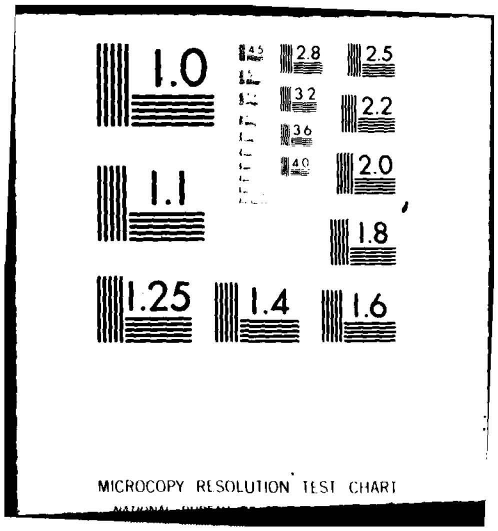
AD-A112 424   CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER --ETC   F/G 9/2
ALOE USERS' AND IMPLEMENTORS' GUIDE.(U)
NOV 81   R MEDINA-MORA, D S NOTKIN
UNCLASSIFIED   CMU-CS-81-145   NL

1 of 1
AD
A112424

END
DATE
FILMED
4-82
DTIC


1.0
2.8
2.5
3.2
2.2
3.6
4.0
2.0
1.1
1.8
1.25
1.4
1.6
MICROCOPY RESOLUTION TEST CHART

CMU-CS-81-145

(12)

AD A112424

ALOE Users' and Implementors' Guide

Raúl Medina-Mora
David S. Notkin

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pa. 15213

November 1981

DEPARTMENT
of
COMPUTER SCIENCE

DTIC
ELECTE
MAR 24 1982
H

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

Carnegie-Mellon University

82 03 23 006

CMU-CS-81-145

12

ALOE Users' and Implementors' Guide

Raúl Medina-Mora
David S. Notkin

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pa. 15213

November 1981

Abstract

A Language Oriented Editor (ALOE) is a tool which supports the construction and manipulation of tree structures while guaranteeing their syntactic correctness. Instantiations of ALOEs are generated from a grammatical description of a language. Trees (*e.g.*, programs) are represented internally by ALOEs as abstract syntax trees that the user manipulates directly. A simple language is provided for mapping this internal representation to a concrete representation for display to the user. Complex ALOEs that need more than just syntactic support may provide underlying action routines, extended commands, and environment specific routines. This guide describes how to use and implement an ALOE.

This work is sponsored in part by the Software Engineering Division of CENTACS/CORADCOM, Fort Monmouth, NJ. and in part by the National University of Mexico.

Table of Contents

DTIC COPY INSPECTED

Accession For
NTIS
[illegible]
~~form~~ 50 Per on file
By
Dist
A

1. Introduction

A Language Oriented Editor (ALOE) is a tool which supports the construction and manipulation of tree structures while guaranteeing their syntactic correctness. Instantiations of ALOEs are generated from a grammatical description of a language. Trees (*e.g.*, programs) are represented internally by ALOEs as abstract syntax trees that the user manipulates directly. A simple language is provided for mapping this internal representation to a concrete representation for display to the user. Complex ALOEs that need more than just syntactic support may provide underlying action routines, extended commands, and environment specific routines. We tend to emphasize the generation of ALOEs for programming languages because they are the most natural application, but ALOEs can be generated for all structures that can be expressed using the grammatical description. This guide describes how to use and implement an ALOE.

The ALOE Generator produces a set of tables that define the language knowledge of every ALOE. These tables are compiled and then loaded with the rest of the editor to form a new ALOE. The ALOEs generated are syntax-directed editors. This means that constructing and modifying programs or structures is done following the abstract syntax of the language (which defines the structure of the language). A large part of what is understood as the syntax of a language is not part of its abstract syntax but of its concrete representation (punctuation marks, keywords, brackets, *etc*). With a syntax-directed editor the concrete representation is no longer important and it is only used for display purposes.

An ALOE can also be visualized as a constructive editor. Language constructs (such as variables, operators, expressions, different kinds of statements) can be added, modified, or removed. The user communicates with ALOE in terms of these language constructs. The programmer constructs his program by inserting templates representing different language constructs and then filling the "holes" of those templates with other templates. Since ALOE knows which constructs are valid at any given point, it allows the programmer to insert a language construct only where it is syntactically correct. For example, instead of typing the character sequence for an **if** statement in C, the programmer calls on the template **if**. The result is the insertion of

```
if (<expression>)
    <statement>
else
    <statement>
```

at the current program position, provided that this construct is syntactically correct in that context. The current program position is advanced to `<expression>` so that it can be similarly expanded. Note that ALOE provides all the necessary keywords, separators, terminators, and all the other "syntactic sugar" required by the language like the parentheses around the `<expression>` construct required by the C language syntax. Problems such as misspelled or non-matching keywords cannot occur because the language constructs are inserted by ALOE and not by the typist.

Internally ALOE represents the program as a *syntax tree.* Each template corresponds to a node of a certain type in the tree. The holes of the template are the offspring of the node. They will be filled in with subtrees representing the expansion of those holes. Thus, the programmer actually constructs and manipulates a program tree without necessarily being aware of it. However, it is desirable for the user to think of the program in terms of the syntax tree instead of the displayed text. To present the programmer the text of his program, ALOE uses an *unparser* that translates the syntax tree into human-readable text. As part of its task this unparser does the pretty-printing of the program. This unparsing process is driven by *unparsing schemes* specified in the grammatical description. These schemes describe the mapping from the abstract syntax to the concrete representation for every language construct. The clear separation of abstract syntax and concrete representation places the emphasis on language constructs and not on specific syntactic details. ALOE relieves the programmer from the worries of the syntax constraints imposed by the language. The user can then concentrate on the contents of the program instead of on its form.

The programmer interacts with ALOE through language commands and editing commands. Language commands are used to insert new templates (*e.g.* the call for an **if** template). Editing commands are used for manipulating the program tree (*e.g.* delete a subtree). ALOE is extensible: Extended commands can be defined to implement other functions of an editing environment and to incorporate language specific functionality. Additionally, action routines can be specified in the grammatical description of the language to be invoked by ALOE in certain situations. They allow the implementor of an ALOE to perform semantic checking, automatically generate program pieces, attach or review information stored in the tree, *etc.* Finally, environment specific routines can be provided. Some simple implementations for these routines are provided for simple ALOEs.

A wide range of ALOEs can be generated. On one end of the scale there is a very simple purely syntactic ALOE that lets the user construct and modify programs or structures. These ALOEs have no action routines, no extended commands, and use the standard implementation of environment specific routines. On the other end of the scale one can generate a full blown software development environment like the Gandalf System [Habermann 79, Notkin 82] which is constructed through implementation of complex action routines, extended commands, and implementation of its own version of the environment specific routines.

ALOEs exist for following languages: the UNIX^{tm} C language [Kernighan 78], the GC language [Feiler 79], Alfa (a functional programming language designed by Habermann [Habermann 80]), the DOD Ada language [Ada 80], Pascal [Jensen 74], Gandalf [Habermann 79], the grammatical description itself, and a large collection of simpler examples like the one described in appendix I. Copies of these grammar descriptions are available from the authors.

The fact that we can generate an ALOE for the grammatical description itself means that the process of creating a grammar for a new language is actually done through an ALOE rather than with a text editor. The language tables are generated with different unparsing schemes.

Chapter 2 of this guide is the ALOE Users' Manual. It explains the use of an ALOE once it has been generated. It defines the editing commands and explains their functionality. It also explains the user interface of every ALOE. The Users' Manual together with the grammatical description of a particular language form the users' manual for that specific language. The former defines the language independent properties and the latter defines the language structure. If extended commands are provided their description must also be added to complete the users' manual.

The rest of the chapters describe how to generate a new ALOE. Chapter 3 describes the ALOE Generator and the grammatical description rules. Chapter 4 describes the implementation of the internal representation. Chapter 5 explains the interface to action routines. Chapter 6 defines the interface to extended commands. Chapter 7 provides an operational definition of the internal structures through a set of routines available to the implementor of an ALOE which can be invoked from action routines or extended commands. Chapter 8 defines the system or environment specific routines whose standard implementation is provided. Any implementation can redefine some or all of these routines. Chapter 9 describes the window manipulation package and how to use it. Chapter 10 describes the use of the display handling package. Chapter 11 explains the process of generating an ALOE once the grammar has been created and the rest of the system has been written. Finally, appendix I gives a detailed example of the implementation of an ALOE for a simple language. This example defines action routines, provides extended commands, and redefines some of the environment specific routines.

Every attempt has been made to ensure that the information contained in this guide is accurate. However, since the ALOE system is under continuing development, the details of this guide may change slightly over time. Serious users of the system will, of course, be notified of these changes. If you find any errors or inaccuracies in this guide, please forward them to the authors.

Throughout this document references are made to some data and executable files. If this system is installed somewhere other than CMU, the UNIX file paths should be substituted for by the paths in that installation. References to "*/usr/gandalf*" should be changed to the *home* directory. The implementation of action routines, extended commands, and environment specific functions can be done in either C or GC (or in any language that is load-compatible with C under UNIX).

2. ALOE Users' Manual

This chapter describes the ALOE user interface. Every ALOE has both *editing commands* and *language* (or *constructive commands*). Editing commands are common to all ALOEs and are described in this chapter. The interface to language commands is also described here, although the actual commands cannot be described here since they are language dependent. Extended commands have the same interface as the editing commands but are specific to a particular ALOE instantiation. Their implementation interface is described in chapter 6.

2.1. The Cursor

An ALOE always displays the current state of development of a tree to the user. The text that is displayed on the screen is generated by unparsing the internal representation using the current *unparsing scheme* (see section 3.1.4). To give the user a clear idea of where in the tree s/he is located, an ALOE displays an *area cursor* which highlights the entire textual expansion of the current subtree. The root of this subtree is called the *current node.* A single character cursor would be ambiguous in this kind of structured context.

Cursor movement does not necessarily follow the textual representation of the tree (as most users are familiar with in text editors) but rather follows the structure of the tree. Depending on the value of the *cursor-follows* mode (see section 2.11) the cursor will follow the internal tree order or the order indicated by the unparsing scheme (in most instances the orders coincide). The result of the move is indicated by highlighting the new cursor.

2.2. Language Commands

Constructive or language commands are the language dependent operator names described in the grammar used to build an instantiation of an ALOE. Essentially, these operators are used to build the structures of the language while the editing commands (described in section 2.4 below) manipulate these structures.

The syntactic correctness of the tree is guaranteed by first defining the types of nodes that can appear in the tree and then defining which node types can appear as offspring of other node types. Nodes are either *terminals* (*i.e.*, leafs) or *non-terminals.* Non-terminals have either an ordered set of offspring or a list of offspring. The types of the offspring are denoted by a *class* which defines the set of terminal and non-terminal operators that are legal as offspring.

These constructive commands can only be invoked when the current node is a meta node. They are invoked by typing the name of the operator or its synonym (see section 2.5 below) followed by a carriage

PRECEDING PAGE BLANK-NOT FILMED

return. Only those characters needed to unambiguously distinguish the operator name must be entered. If the desired operator belongs to the current class, the ALOE inserts a subtree of this operator type in the place of the meta node, filling the offspring with the corresponding meta nodes. The cursor is then placed at the first meta node in the subtree. If the operator is a terminal operator then the subtree consists of a single node and the cursor is moved to the next available meta node.

2.3. Automatic Application

When a meta node is created by a constructive command, it may be possible to automatically replace it with additional structure. The automatic application is done if the meta node is in a class with only one operator, which is a non-terminal. If the meta node is "non-visible" (as defined in its parent unparsing scheme, see section 3.1.4) then the automatic application is not done.

2.4. Editing Commands

Editing commands are common to all ALOEs. These commands are invoked by typing a dot (".") followed by the name of the command and a carriage return. Only enough characters to designate the command unambiguously need be entered. For the common commands, one character usually is sufficient. All editing commands also have synonyms defined which are entered without either the dot or the carriage return and are, in general, control characters. Some commands require arguments, like the name of a file, the name of a tree, *etc.* These arguments can be given directly after the command or in response to ALOE prompts. When a prompt is given a default value is shown enclosed in square brackets. A simple carriage return indicates that the default value should be used. Otherwise, the value entered is used as the argument.

The editing commands for an ALOE are described next. The synonyms for each command are shown in parentheses following the command name. The "$" symbol in synonyms stands for the *<escape>* character.

2.4.1. Cursor Movement

Cursor-in

._IN (<cursor-pad-down> or $<)
Moves the cursor into the first legal offspring of the current node according to current unparsing scheme. **Cursor-in** does nothing if applied at a terminal or non-visible node.

Cursor-out

._OUT (<cursor-pad-up> or $;)
Moves the cursor to the parent of the current node.

Cursor-next

._NEXT (<cursor-pad-right> or $=)

Moves the cursor to the next sibling of the current node if one is defined according to the current unparsing scheme and the setting of the *cursor-follows* mode (see section 2.11). If no sibling is defined, the cursor is then moved to the next sibling of the parent of the current node, recursively. If the current node is the last in the tree (as defined in pre-order) then the command has no effect.

Cursor-previous

._PREVIOUS (<cursor-pad-left> or $>)

Moves the cursor to the previous sibling of the current node if one is defined according to the current unparsing scheme and the setting of the *cursor-follows* mode. If the current node is the leftmost node then the cursor is moved to the previous sibling of the parent of the current node. If the current node is the leftmost node in the tree (as defined in pre-order) then the command has no effect.

Cursor-home

._HOME (<cursor-pad-home> or $?)

If the current node is not the root of the current window, **cursor-home** moves the cursor there. Otherwise, it moves the cursor to the root of the previous context window. See section 9.2 for more details on windows.

Cursor-back

.BACK **(↑b)**

Moves the cursor back to its previous position, provided that the last command was a cursor moving command.

Find

.FIND <string> **(↑f)**

Searches the tree for a matching variable name, constant name, operator synonym, or operator name. The search is restricted to the current window. If no string is given one is prompted for. If a carriage return (<cr>) is typed for the string prompt the string specified in the previous .FIND or .GLOBALFIND is reused.

Global Find

.GLOBALFIND <string> **(↑g)**

Searches the current tree for a matching variable name, constant name, operator synonym, or operator name. Extends the search beyond the current window. If no string is given one is prompted for. If a <cr> is typed to the prompt then the string specified in the previous search (with .FIND or with .GLOBALFIND) is reused.

Numerical Arguments for Cursor Movement

.<number>

The explicit cursor moving commands (**cursor-in**, **cursor-out**, **cursor-next**, **cursor-previous**, and **cursor-home**) have an optional parameter that precedes them. The numerical argument indicates how many applications of the given command should be made. The argument is not a command in that it cannot be used alone.

2.4.2. Help Information

All help information available through these commands is displayed in the help window (see section 2.13).

Operator Help

.HELP **(↑x?)**

If the current node is a meta node, .HELP displays the list of applicable language commands (and their synonyms). Otherwise, the list of editing commands is displayed.

Command Help

.? (.?)

Displays the list of editing commands (and their synonyms).

2.4.3. *Tree Manipulation*

2.4.3.1. Clipping

Clip Subtree

.CLIP <tree-name> **(↑k)**

Clips current subtree into a named tree which is kept in the clipped area separate from the main tree. The name of the tree can be specified following the command or it will be prompted for.

Insert Subtree

.INSERT <tree-name> **(↑x↑i)**

Inserts a clipped subtree at the current node (which must be a meta node) provided that the root operator of the subtree is legal in this position. If no tree name is specified one is prompted for.

2.4.3.2. List Manipulation

Extend List

.EXTEND **(↑e)**

Extends a list with a new meta node. If the current node is a list node (variable arity node) then an element is created at the beginning of the list. If the current node is a member of a list then the meta node is inserted immediately after it.

Extend List Backwards

.BEXTEND **(↑x↑b)**

If the current node is a member of a list, it places a meta node immediately before the node. It is not applicable anywhere else.

Prepend to List

.PREPEND **(↑x↑a)**

If the current node is a member of a list, it places a meta node at the beginning of the list.

Append to List

.APPEND **(↑x↑e)**

If the current node is a member of a list, it places a meta node at the end of the list.

2.4.3.3. General Manipulation

Delete

.DELETE **(↑d)**

Deletes the current subtree. If the subtree is an element of a fixed arity node, then a meta node is inserted in its place. If the subtree is an element of a list, the element is removed completely from the list.

Replace

.REPLACE **(↑r)**

Deletes the current subtree. If the subtree is an element of a fixed arity node, then a meta node is inserted in its place. If the subtree is an element of a list, the element is replaced by a meta node of the appropriate class.

Nest

.NEST <operator name> **(↑n)**

Takes the current subtree and nests it into a subtree that will have the operator as root operator. The operator name can be given following the command or it will be prompted for. A nesting that would result in an invalid tree is not permitted. If the new subtree has more that one offspring, it finds the first match (independent of unparsing scheme) for the current subtree in the new subtree.

Transform

.TRANSFORM <operator name> **(↑t)**

Transforms the operator of the current node to the desired one. For the transformation to succeed, the new operator must be in the same class as the old one and the respective offspring must also match exactly.

2.4.4. Input/Output

Read Program

.READPROG <file-name> **(↑x↑r)**

Reads a tree from a file. Checks that the file contains a valid tree. Replaces the current tree with the new tree. Checks with the user if the current tree has not been saved. The file name can be given after the command or given to the ALOE prompt.

Load Tree

.LOADTREE <file-name> **(↑x↑v)**

Loads a tree from a file into a clipped area. A clipped window is assigned to it with the name of the window taken from the file name (see chapter 9 for more information on windows). The name of the file can be given after the command or given to the ALOE prompt.

Write Tree

.WRITE **(↑x↑w)**

Writes a tree into a file in tree form. The default prompt is the file name given at invocation of ALOE (see section 2.15 below).

Unparse into File

.UNPARSE *<file-name>* ***(↑x↑t)***

Unparses the tree into a text file. The file name can be given after the command or to the ALOE prompt. Useful for producing printouts. Note that this command differs from .WRITE only in the form the written file takes.

2.4.5. Exit ALOE

Quit and Save

.QUIT **(↑x↑f)**

Saves the current tree in a file in tree format and leaves ALOE. It uses the file name given at invocation.

Cancel

.CANCEL **(↑c)**

Leaves ALOE. If the tree has been changed since the last .WRITE command, the user is warned and given a chance to abort the command.

2.4.6. Display Manipulation

Display Tree

.DISPLAY **(↑l)**

The screen is cleared and redisplayed. Useful when operating system messages or other such noise gets displayed in the screen.

Window Down

.WDOWN **(↑w↑n)**

Scrolls the tree window down by one half of the window length.

Window Left

.WLEFT **(↑w↑a)**

Scrolls the tree window left by one third of the window width.

Window Right

.WRIGHT **(↑w↑e)**

Scrolls the tree window right by one third of the window width.

Window Up

.WUP **(↑w↑p)**

Scrolls the tree window up by one half of the window length.

Select Window

.WINDOW <window-name> **(↑w↑w)**

The selected window is displayed on the screen. The window name can be given after the command or to the ALOE prompt. The window must be a tree or a clipped window (see chapter 9). If it is a tree window, the appropriate context switch takes place as if the cursor had been moved there explicitly.

2.4.7. Other Commands

Fork a UNIX Shell

.! **(↑x!)**

Calls the UNIX shell from ALOE.

Edit

.EDIT **(↑x↑t)**

Invokes EMACS [Gosling 81a], a text editor, to edit a constant, long constant, or text node. Upon return the screen is updated to incorporate the edited string.

Set Mode

.MODE (↑x↑m)
Sets the mode to the new value. ALOE first prompts for the name of the mode and then for the new value of the mode.

Set Unparsing Scheme

.SCHEME <scheme-number> (↑xU)
This is a command to let the user change the current unparsing scheme. Takes as argument the number of the unparsing scheme. The scheme number can be specified after the command or given to the ALOE prompt. If the argument is out of range (larger that the largest defined unparsing scheme) then scheme zero is used.

2.5. Synonyms

Synonyms are defined as alternative names by which commands can be invoked. For language commands, the synonyms are specified in the grammar (see section 3.1.6). Their values may be found by invoking the .HELP command and seeing what parenthesized string follows the operator name. If there are duplicate synonyms for operators in the same class, only the first one in the class will be matched. Editing command synonyms can be found by executing the .? command. The synonym follows the command name and is shown in parentheses.

2.6. Extended Commands

Extended commands are commands added to the basic set of ALOE editing commands to implement language or environment specific operations. Their user interface is the same as that of editing commands. The names and synonyms of extended commands should be chosen so that they do not conflict with those of the basic editing commands. For a description of the implementation interface for extended commands, see chapter 6.

2.7. EMPTY as Command

Many languages contain operators which are optional. ALOE supports optional operators via the operator name EMPTY. Indeed, any operator name beginning with EMPTY is treated specially by the command interpreter (as is described below). Multiple EMPTY operators are provided for cases where different formatting effects are desired for different optional operators.

2.8. Carriage Return as Command

A <cr> is interpreted as a command when applied at a meta node. Otherwise it has no meaning. If the meta node is an element of a list, the <cr> indicates that the user is "done" entering the list elements. Hence, the meta node is deleted and the cursor is moved to the next meta node. If, instead, the class of the current meta node contains an EMPTY operator, then the EMPTY operator is applied. This will match any operator that begins with EMPTY in the class. If neither of these cases applies then the cursor is moved to the next meta node.

2.9. Command Terminators

Every constructive command can be terminated by either a <cr> or by a line-feed (<lf>). If a <lf> is used then the cursor will be left at the newly created node instead of at the first available meta node.

2.10. Multiple Commands in a Line

Several commands can be issued in a single line. They must be separated by spaces. The screen will be updated only after all commands have been processed.

2.11. ALOE Modes

ALOE behavior varies slightly depending on the values of several modes. Current modes include *cursor-follows* mode, *novice* mode, *editmode*, and *show-windows* mode. Cursor-follows mode indicates whether the cursor follows the tree structure or the current unparsing scheme. In either case nodes that are not "visible" (*i.e.*, are not referenced in the current unparsing scheme) are not visited. The default is to follow the tree structure. The unparsing scheme may specify that the offspring of a fixed arity node be unparsed in a different order than that of the abstract syntax specification (see section 3.1.4). Novice mode indicates that the help menu will always be displayed after every command. The default value for this mode is **off**. Editmode indicates that the cursor will not be moved to the next meta node available in the tree after construction. The default value for this mode is **off** unless a previously constructed tree is used on invocation (see section 2.15 below). Show-windows mode indicates if the context-window window should be displayed (see section 9.1). Its default value is **off**. This mode is only useful in ALOEs where different contexts are displayed using different overlaying windows.

2.12. Error Reporting Interface

Action routines (see chapter 5) and extended commands (see chapter 6) buffer messages of three types: errors, warnings, and messages. Upon return from action routines or extended commands, ALOE will display the messages to the user one by one, highlighting the node where the message occurred. After seeing any message the user can decide to stop the reporting process.

2.13. Screen Organization

In every ALOE instantiation the screen is divided into different *windows*. The location and size of these windows can be redefined by the user by modifying a window definition file (see chapter 9 for more information).

The main tree is displayed using a *root* window. Different contexts of the tree can be displayed using different overlaying windows from a pool of tree windows. Clipped trees are displayed in *clipped* windows normally placed in the bottom half of the tree windows. Errors reported from action routines (see chapter 5) are displayed in an *error* window that normally overlays the tree windows.

Help information is displayed in a *help* window normally placed at the right of the screen. The *status* window is a one line window normally near the bottom of the screen. It displays status information including the name of the current tree window, the operator name of the current node, the class it belongs to, and the current values of the different modes: "Tree" or "Scheme" for *cursor-follows* mode, "Novice" or "Expert" for *novice* mode and "Editing" or "Constructing" for *editmode*. The value of show-windows mode is not shown in the status window but is clear from the presence or absence of the context window itself.

2.14. Checkpointing

ALOE will write out a checkpoint tree file (extension "*CHK.tr*") after thirty tree modifying commands. See section 8.4 for a description of how to change the frequency of checkpointing.

2.15. Parameters to an ALOE

An invocation of an ALOE may have both a variety of switches and also two optional file names. Any switches (which may be in either upper or lower case) and their parameters must precede the file names. The meaning of each switch and of each filename is described next.

2.15.1. Unparsing Scheme: the -u Switch

Using -u<number> sets the initial unparsing scheme used by ALOE to scheme <number>. The default is scheme zero. If the argument is out of range, scheme zero is used.

2.15.2. Parse: the -p Switch

A parser can be interfaced with an ALOE to be able to edit programs created previously with text editors. The -p<file-name> switch indicates that the initial tree is to be created by parsing a text program contained in <file-name>. ALOE invokes the parser by calling the routine Gparse to perform the parsing. Currently, only a parser for the GC language [Feiler 79] has been written. Details of the parsing interface are available from the authors and are not included in this guide.

2.15.3. Tracing Action Calls: the -s Switch

The -s switch causes a trace of each action call invocation to be displayed on the screen. The operator on which the call is being made is listed and the type of the action call is given. If, instead, -s<filename> is given, a trace of all the action calls is listed in the file <filename>. This is often useful for debugging the action routines.

2.15.4. An ALOE as a Driven Process: the -r Switch

An ALOE can be driven as a separate process using the *pipe* mechanism in UNIX. The -r<process-id> switch specifies the process id number of the parent process to which ALOE will send interrupt signals whenever it is ready to receive input.

2.15.5. File Names

The first file listed indicates a tree file to be read in as the initial tree. ALOE checks the name of the language and the version of the ALOE that previously wrote the tree. Editmode (see section 2.11 above) is initialized to **on**. If no file name is given the ALOE starts by providing a subtree with the root operator as its root and editmode is initialized to **off**.

If a second file is listed it indicates that the invocation of the ALOE is not interactive. Instead, the indicated initial tree file is unparsed (by scheme zero or as indicated by the -u switch) and the results are placed into the second file listed.

3. The ALOE Generator

In order to instantiate an ALOE, a description of the grammar for the language to be edited must be created. To do this, an ALOE generator is provided. The generator is an ALOE instantiation: The grammar can be expressed in terms of itself and therefore an ALOE can be generated for it. Grammars are then created and modified using this ALOE (see section 11.1.1). Below, the general form of the grammar description is explained. Then, a grammar for a simple language example is given. This example is dealt with in detail in appendix I.

3.1. The Grammar Description

It is useful to have a reasonable understanding of what the ALOE that is generated provides before actually attempting to generate one. Hence, it is important to read the earlier chapters of this guide that describe language operators and such before one reads this chapter in detail.

The description consists of several conceptual pieces which are interleaved in the actual description. These pieces are:

- The name of the language.
- The description of the abstract syntax of the language.
- The root operator.
- The precedence of each non-terminal operator.
- The unparsing schemes that describe how the (internal) tree form is displayed on the screen. In other words, this describes (potentially many) mappings from the abstract syntax to the concrete representation.
- The action routines associated with each operator of the language. These are needed to expand on the syntactic capabilities of the basic ALOE. For a detailed explanation of action routines, see chapter 5.
- The synonyms available for each operator.
- Whether or not the corresponding node of a non-terminal operator should be a root of a separately stored tree.

PRECEDING PAGE BLANK-NOT FILMED

3.1.1. Language Name

Each grammar has a language name associated with it. This is used to mark all tree files that are created and stored with an ALOE. The ALOE checks this name and does not attempt to edit a tree created by an ALOE for another language. The implementor of an ALOE should be careful to change the grammar name whenever a structural change to the grammar is made. Otherwise, the ALOE may attempt to edit trees that are really of a different language.

3.1.2. The Language Description

The description of the syntactically correct sentences in the language is done in a two level structure. One level provides a description of each operator of the language. There are two possible descriptions of this type: non-terminals (nodes that represent either an ordered set of offspring or a list of offspring) and terminals (nodes which represent leaves of the tree). The second level of the description lists the classes of the language. Classes represent the set of legal operators that can be created in a place of an unfilled-in offspring (referred to as a "meta node"). So, a good way to look at the syntactic description is that the sentences are AND/OR structures where the operators provide the AND function and the classes provide the OR function.

There are several types of terminal nodes:

- *Statics* are nodes that represent some concrete piece of the language, like the name of a type.
- *Constants* are nodes containing ASCII values other than blanks. The values are prompted for when the node is created. Used for integer or character constants.
- *Long constants* are constant nodes that may contain blanks. Used for comments, strings with blanks.
- *Text* nodes are constants that are entered and edited with EMACS [Gosling 81a], a text editor external to ALOE. Used for longer pieces of documentation.
- *Variable* nodes are automatically entered into a simple name table structure. Information on how to implement more general symbol table manipulation methods is described in section 8.1.

3.1.3. Precedence

Non-terminal operators might have precedence values (integers) associated with them. These are used to provide automatic parentheses while unparsing. They are optional (default value of precedence is highest precedence). It should be clear, however, that the precedence values are only necessary for display purposes, since the internal tree is always unambiguous.

3.1.4. The Unparsing Schemes

The internal form that is created by the ALOE to represent the sentences of the language is kept in a tree form (both for manipulation by the ALOE and storage in the file system). In order to make the tree readable by the user, a set of unparsing schemes must be provided in the language grammar.

Each non-terminal and terminal operator must have at least one unparsing scheme associated with it. Each unparsing scheme is a string which has descriptions of the text to be used, the syntactic sugar to be used, and the way the actual object (if there is one) is to be formatted. The text given in the unparsing scheme is displayed as is. Only sequences starting with the "@" or "%" character are treated differently. The formatting commands available in unparsing schemes are:

@n	Insert a newline in the output.
@t or @>	Insert four spaces in the output.
@<	Go back four characters (stopping at the beginning of the line).
@+	Increase the indentation level (to take effect at the next "@n"). Every indentation is four spaces.
@-	Decrease the indentation level (to take effect at the next "@n").
@l	Flush left (start next piece of output at left margin of current line).
@h	Go back one character (provided it is not at the beginning of a line).
@b	Go back to previous line (undo "@n").
@u<n>	Change the current unparsing scheme to <n>. Push the current scheme index on a one-level stack.
@u	Reset unparsing scheme to value of the one-level stack.
@p<n>	Push marker <n> onto stack. Markers are used to "remember" column positions for formatting. They are specially useful when the desired formatting depends on the size of identifiers. First marker is numbered zero.
@r<n>	Pop marker <n>.
@g<n>	Get marker <n>. Moves the unparsing cursor to the column position specified by the marker.
@@	Display an "@" character.

`@%`	Display a "%" character.

The way the objects of the nodes are displayed is different for terminals and non-terminals. For terminals, the following unparsing commands are available:

`@c`	Display value of constant, long constant, or text
`@s`	Display variable name from symbol table.

The unparsing commands available for non-terminals are:

`@<n>` Unparse the <n>th offspring recursively. Used only for fixed arity nodes where the offspring are numbered from one on up. For example, "`while (@1)@+@n@2@-@n`" specifies that the node should be unparsed starting with the string "`while (`" followed by a recursive invocation of the unparser on the first offspring, a "`)`", a line break, the unparsing of the second offspring indented one level, and another line break. The order in which the offspring are unparsed can be different from the one specified in the abstract syntax. The "`n`" in "`<n>`" refers to the abstract syntax specification order. Finally, nodes can be hidden (made "non-visible") by simply omitting them in the unparsing scheme.

`<pr>@0<t>[@q<po>][@e<s>]`
Unparse the list node. Used only for non-terminals that are list nodes. The "`@0`" indicates that each element of the list should be unparsed in order. The "`<pr>`" is the prelude string that should be printed before the list is unparsed. The string "`<t>`" is used to separate list elements "`(<t>`" is terminated by either the following "`@q`" or the end of the unparsing scheme). The string "`<po>`" is the postfix that is printed after the list is unparsed. The optional "`@e`" indicates how the list should be unparsed if it is empty (*i.e.*, has no current elements). All of the strings may contain text and other unparsing commands. The parts in square brackets are optional. If no empty specification is given then nothing will be unparsed when the list is empty. For example, the scheme "`versions:@+@n@0; @n@q@-@nend@e<no versions>`" specifies that the list should be unparsed starting with the string "`versions:`", a line break and then the elements of the list separated by a "`;`" and a new line. The list should be terminated by a new line and the word "`end`" aligned with "[versions:]". If the list is empty then it should only unparse the string "`<no versions>`".

`@x` Used only for non-terminals that have filenodes associated with them and indicates that the subtree is not "visible". More on filenodes appears in section 4.4.

`@z` Used in conjunction with "`@x`" to specify the place where the name of the filenode should be placed.

All the letters after the "`@`" in unparsing commands may be either upper or lower case. Additionally, with one exception, anywhere an "`@`" can occur a "`%`" can also be used. The exception is that in fixed arity operators, "`@<n>`" means that the node should be unparsed and visited (*i.e.*, you can stop the cursor there), while "`%<n>`" means to unparse but not to visit the node. In the case of lists it means that no element of the list can be visited.

As said before each operator may have multiple unparsing schemes associated with it. The default scheme is zero (this may be altered on invocation of ALOE by the use of the -u switch (see section 2.15) or with the .SCHEME command (see section 2.4.7)). The "@u<x>" command may be used to modify the current scheme number from within schemes.

3.1.5. Action Routines

Each operator (terminal or non-terminal) has an optional action routine associated with it. This routine is called when a node of this type is created, deleted, exited, *etc*. Just the name of the action routine need be listed. Each routine takes two parameters: One, the actual node on which the call is being made; and two, the type of operation that caused the call to be made. These routines are used to expand on the syntactic capabilities of the basic ALOE, including semantic checking, automatic generation of program pieces, communication with other parts of the editing environment such as access control, code generation, *etc*. Complete details on the action routine interface can be found in chapter 5.

3.1.6. Synonyms

Each operator may have a synonym associated with it. This is so that, for instance, it is possible to invoke an addition by typing "+" instead of "PLUS".

3.2. An Example of the Grammar Description

We will illustrate the grammar description using the grammar for the example language of appendix I. The grammar is displayed to the user in four separate pieces. Each one is described in a separate section below.

3.2.1. Language Name and Root

Each grammar description is associated with a language name. This is stored with the tree so that ALOEs can distinguish among trees written with different ALOE instantiations. It is the responsibility of the grammar writer to ensure that there are no name conflicts among ALOEs for different languages and that when incompatible changes are made to a grammar that the name is also changed.

Additionally, the grammar designates one of the non-terminals as the root of the grammar. This is needed in order to start the construction process in the appropriate place.

These pieces of the grammar are displayed to the user as:

```
Language Name: INTERP
Root Operator: PROGRAM
```

3.2.2. Terminal Productions

The terminal productions comprise the next piece of the grammar that is displayed to the user. The description of each terminal symbol consists of five parts, each of which is explained below. The character "|" is used to separate the different parts of the production.

3.2.2.1. Terminal Operator Name

The terminal operator's name is the first part displayed. The generator will convert all operator names to all upper case. The operator names that start with the characters EMPTY are treated specially, see section 2.8.

3.2.2.2. Terminal Type

The next part of the terminal production is the type of the terminal. Statics are displayed as {s}, constants as {c}, long constants as {a}, text nodes as {t}, and variables as {v}.

3.2.2.3. Unparsing Schemes

The next part defines the unparsing schemes to be used to map the internal abstract syntax to the external concrete representation. It is probably the most complicated part of the grammar description. Schemes are numbered starting at zero (the default scheme at unparsing time). The definition of the unparsing schemes for a terminal (or indeed, a non-terminal) consists of a list of pairs: The first element of each pair is a list of numbers and ranges indicating which unparsing schemes are defined by the pair; and the second element of the pair is a string which is the actual unparsing scheme. These ranges should not intersect. All schemes whose number is less than the highest scheme number defined but not defined in any of the lists will have a default "**error**" scheme associated with them. This is helpful for debugging purposes.

3.2.2.4. Action Routines

The name of the action routine associated with the operator is specified after the unparsing schemes. If there is no action routine, then "<none>" is displayed.

3.2.2.5. Synonyms

The synonym is defined after the action routine. If there is no synonym for a terminal operator, then "<none>" is displayed.

3.2.2.6. Terminal Operators of Example

```
{
LOOPVAR =       {v}
              | (0) "@s"
              | action: <none>
              | synonym: "'" ;
INT =           {c}
              | (0) "@c"
              | action: aINT
              | synonym: "#" ;
EMPTYSTEP =     {s}
              | (0) "1"
              | action: <none>
              | synonym: <none> ;
}
```

3.2.3. Non-terminal Productions

Most of the parts of the production for a non-terminal operator are similar to that of a terminal. For the non-terminal, there are six pieces that the user must construct.

3.2.3.1. Non-terminal Operator Name

This is defined as in the terminal name described above.

3.2.3.2. Offspring

There are two types of non-terminals -- fixed arity nodes and variable arity nodes (*i.e.*, lists). Fixed arity nodes are described by listing the (ordered) set of offspring after the name of the operator. Each son is represented by the name of the class that specifies the legal operators for that field. Variable arity nodes are described by listing the name of the class from which elements of the list must be selected. This is displayed by the enclosing of the class name in angle brackets.

3.2.3.3. Unparsing Schemes

These are defined identically to the ones previously described in the section for terminals.

3.2.3.4. Synonyms

These too are identical to the synonyms for terminals.

3.2.3.5. Precedence

The next part of the description of non-terminals is the precedence. This is displayed as the precedence value or "<none>" if no precedence is assigned.

3.2.3.6. Filenode

The final part of the description notes whether or not the non-terminal is a standard non-terminal or is a filenode non-terminal (or FNONTERMINAL). An FNONTERMINAL is a non-terminal that has its subtree stored in a separate file (for storage and checkpointing purposes). This is essentially a binary flag that is displayed as either "`filenode`" or "`non-filenode`". The Root operator is automatically made an FNONTERMINAL by the generator. More on filenodes can be found in section 4.4.

3.2.3.7. Non-terminal Operators of Example

```
{
PROGRAM =      stmts
             | (0) "@1"
             | action: <none>
             | synonym: <none>
             | precedence: <none>
             | Filenode;
PRINT =        <exp>
             | (0) "print @0,"
             | action: <none>
             | synonym: <none>
             | precedence: <none>
             | Non-filenode;
FOR =          loopvar exp exp stepexp stmts
             | (0) "for @1 = @2 to @3 step @4@+@n@5@-"
               (1) "for (@1 = @2; %1 <= @3; %1 =+ @4)@+@n@5@-"
             | action: <none>
             | synonym: <none>
             | precedence: <none>
             | Non-filenode;
PLUS =         exp exp
             | (0) "@1 + @2"
             | action: <none>
             | synonym: "+"
             | precedence: 1
             | Non-filenode;
TIMES =        exp exp
             | (0) "@1 * @2"
             | action: <none>
             | synonym: "*"
             | precedence: 2
             | Non-filenode;
STMTS =       <stmt>
             | (0) "@0@n"
             | action: <none>
             | synonym: <none>
             | precedence: <none>
             | Non-filenode;
}
```

3.2.4. Classes

The last part of the formal description lists the classes of the grammar. Classes are the OR part of the AND/OR structure. Each class name is followed by a list of names of terminal and non-terminal operators. These are then the only legal replacements for the meta node during construction or editing.

3.2.4.1. Classes of Example

```
{
stmts        =      STMTS ;
exp          =      INT LOOPVAR PLUS TIMES ;
loopvar      =      LOOPVAR ;
stepexp      =      INT PLUS TIMES EMPTYSTEP ;
stmt         =      PRINT FOR ;
}
```

4. Internal Representation

This chapter describes the internal tree representation used and manipulated by ALOE. It only need be understood by users who are creating an ALOE that uses action routines or extended commands which need to access and manipulate these structures.

4.1. Generic Node Reference

Tree nodes are linked together to form trees. The first two words of every node type are common. This allows for generic access to every node regardless of type. The opt field of every node is used to access the print name, the arity, and the type of the node (see chapter 7 for details on routines which provide this information). The father field is a pointer to the parent of the node. This generic node definition used to initially access any node is (the other fields in the definition are discussed later):

```
struct tnode{
    int opt;                        /* index into operator table   */
    struct tnode *fathptr;          /* ptr to father node          */
    int status;                     /* current status of node      */
    struct tnode *sonptr[1];        /* ptrs to sons: real size     */
    };                              /*  determined upon allocation */
```

4.2. Non-Terminal Nodes

The two different kinds of non-terminal nodes are each represented differently. These two representations, however, share the other two fields listed in the tnode definition above. The status field contains status information which can be accessed directly or by using the support described in section 4.5. The differences in the representation are based on how each uses the sonptr field.

4.2.1. Fixed Arity Nodes

If the tnode is representing a node with a fixed number of offspring, the sonptr array is allocated to the needed size (*i.e.*, one entry for each offspring). Each offspring in the array is ordered as it is listed in the abstract syntax specification of the grammar description. No sonptr entry is ever NIL: If an offspring has not yet been expanded, a meta node (tnodem) is pointed to by the sonptr entry.

PRECEDING PAGE BLANK-NOT FILMED

4.2.2. Variable Arity Nodes

If the tnode is representing a node with a variable number of offspring, the sonptr array is one element long. This entry is either NIL (indicating an empty list) or else it is a pointer to the first element of a linked list of structure of type listnode (often referred to as header). The listnode type is defined as:

```
struct listnode{
    struct listnode *nextptr;         /* ptr to next element in list */
    struct tnode *node;               /* pointer to the node itself  */
    };
```

The last element of the list has a NIL nextptr field. As in the sonptr field of fixed arity nodes, the node field of the listnode is never NIL, but may point to a meta node.

4.3. Terminal Nodes

All the terminal nodes share the opt, father, and status fields of the generic tnode definition. The differences are in the later fields of the terminals.

4.3.1. Node for Constants

The representation of constants, long constants, and text nodes extend the first three fields of the generic tnode with a pointer to a character string which contains the value of the constant or text.

```
struct tnodec{
    int opt;                          /* index to operator table      */
    struct tnode *fathptr;            /* ptr to father node           */
    int status;                       /* current status of node       */
    char *ctname;                     /* value of constant            */
    };
```

4.3.2. Node for Statics

The representation of static nodes has only the basic information for the tnode.

```
struct tnodes{
    int opt;                          /* index to operator table      */
    struct tnode *fathptr;            /* ptr to father node           */
    int status;                       /* current status of node       */
    };
```

4.3.3. Node for Variables

The representation of variable nodes has two extra fields which are manipulated by routines involved with the symbol table.

```
struct tnodev{
    int opt;                        /* index to operator table      */
    struct tnode *fathptr;          /* ptr to father node           */
    int status;                     /* current status of node       */
    struct symtabentry *key;        /* key to symbol table          */
    struct tnodev *namethread;      /* thread through commonly      */
    };                              /*      named variables         */
```

4.3.4. Node for Metas

The representation for meta nodes is simply extended with a field indicating the print name of the meta.

```
struct tnodem{
    int opt;                        /* index to operator table      */
    struct tnode *fathptr;          /* ptr to father node           */
    int status;                     /* current status of node       */
    char *metaname;                 /* print name of meta           */
    };
```

4.4. Filenode

A filenode is used to keep information about a file that contains a subtree. It can also have a symbol table associated with it. In the grammar any non-terminal operator can be specified as a filenode, which means that every occurrence of the operator is a root of a subtree that is stored in a separate file. This is the way to achieve the separation of programs and data bases into smaller pieces. Unparsing schemes can be used to "hide" the subtrees that are pointed to by the filenode (see section 3.2.3.6). Also see chapter 7 for the routines that handle the manipulation of filenodes.

The representation of filenodes shares the first two fields of the tnode definition as described earlier.

```
struct tnodef{
    int opt;                        /* index into operator table      */
    struct tnode *fathptr;          /* ptr to father node             */
    char *Myname;                   /* printname of tnodef            */
    char *filepath;                 /* path to file containing subtree */
    struct symboltable *stable;     /* pointer to symbol table        */
    char Status;                    /* dirty and other status values  */
    int stype;                      /* copy of opt field of the son   */
    struct tnode *sonroot;          /* NIL if subtree not loaded      */
    };
```

The `Myname` field is used by the unparser as the value to print when "`@z`" is found in an unparsing scheme (see section 3.1.4). The `filepath` field names the file that the subtree is stored in. The `stable` field is (if it is non-NIL) a pointer to the symbol table to be used in this context (*i.e.*, subtree). The `Status` field is used by the filenode manipulation routines and should not be touched. The `stype` field indicates the `opt` value of the offspring of the filenode (*i.e.*, its FNONTERMINAL). And sonroot points to the subtree if it is in core and is NIL otherwise.

4.5. Accessing the Status

ALOE allows one word for status information. Internally this word is partitioned into several pieces. To access these pieces a tnode can be casted into a tnstat. Action routines use the status field to record the state of subtrees. The first part of the status (the unpidx field) is used by ALOE to record the value of the current unparsing scheme used for this node.

```
struct tnstat {
    int opt;                        /* index into operator table    */
    struct tnode *fathptr;          /* ptr to father node           */
    char unpidx;                    /* Used by editor: DO NOT TOUCH */
    char actstat;                   /* OK, NOTOK, etc.              */
    short int count;                /* How many sons are OK?        */
    };
```

The actstat and count fields may be used in any way the implementor of an ALOE sees fit. See section 7.7 for some routines to help manipulate the status field.

5. Action Routine Interface

An ALOE will call action routines (when they are defined in its tables) in a variety of situations. These calls permit the implementor of an ALOE to attach extra information to the tree, perform non-syntactic checking, do windowing of the user screen, communicate with other parts of the editing environment such as access control or code generation, automatically generate program pieces that can be inferred from others, and much more. Indeed, the limits of what can be done by action routine calls have not been explored completely, but certainly do promise a wide variety of possibilities.

5.1. The Calling Sequence

When an action call is made, two parameters are always passed -- the node where the action is occurring and the type of the action call. The procedure specification for C procedures that are action routines should be:

```
actionROUTINE(thisnode, actkind)
struct tnode *thisnode;
int actkind;
{
        /* implementation code */
}
```

The values for the enumerated type represented by `actkind` are included in the ALOE Library specification file "*/usr/gandalf/include/ALOELIB.h*". The description of when each type of call is made follows. Note that action calls are never made in clipped areas.

5.2. The Kinds of Action Calls

The following sections describe in detail the different kinds of action routine calls. If the construction process of the tree would be static (*i.e.* once the tree is created it is never modified) then the only action call needed would be on CREATE. Indeed this is the type of interface found in parser generators that include semantic routines.

ALOEs are interactive systems with the trees and the display constantly changing. There are several types of changes to the tree and the calls on DELETE, INSERT, EDIT, NEST, TDELETE and TRANSFORM are provided to let action routines handle their effects. Action routines get control of the window manipulation (see chapter 9) through the calls on FAILUP and FAILDOWN. Communication with other pieces of the environment (such as access control, code generation, debugging, *etc.*) is achieved through the calls on ENTRY and EXIT.

5.2.1. CREATE

An action call on CREATE is made when a node is created as the result of a constructive command. A separate call on ENTRY (see below) is *not* made in this case, so the action routine should behave as an implicit ENTRY call. After the action call to create, ALOE will attempt to automatically apply productions to all metas in the subtree of this node (see section 2.3). The automatic application will cause additional calls on CREATE, but those are followed by calls to EXIT of those nodes, leaving the cursor at the first created node. The call to the action routine on non-terminal nodes can be used to automatically generate some fields (defaults, *etc.*). The CREATE action can be aborted and ALOE will then *undo* the operator application and put the meta node back in its place. See section 5.3.1 below.

5.2.2. EXIT

Action calls on EXIT are made while leaving a subtree on a **cursor-out**, .FIND, .GLOBALFIND, **cursor-next**, **cursor-previous**, **cursor-home**, and .WINDOW commands and while looking for meta nodes to be expanded. Also called on nodes exited on a .BACK command or when the cursor is redirected due to the error reporting mechanism (see section 5.3.5).

5.2.3. ENTRY

An action call on ENTRY is made before going into a subtree on an **cursor-in**, **cursor-next**, **cursor-previous**, .FIND, or .GLOBALFIND commands and while looking for meta nodes to be expanded. In addition, ENTRY is called on nodes entered on a .BACK command and when the cursor is moved to a different location due to the error reporting mechanism in ALOE (see section 5.3.5 below).

5.2.4. DELETE

An action call on DELETE is made before doing the deletion on a .DELETE command. The call is performed for the root of the deleted subtree, not for any nodes contained in it. The action routine should behave as an implicit EXIT of the node to be deleted. The DELETE action can also be aborted in which case the ALOE will not delete the subtree.

5.2.5. INSERT

An action call on INSERT is made after an .INSERT command. It should behave as an implicit ENTRY on the root of the inserted subtree. The INSERT call is made on the root of the inserted subtree. Then, as each node in the subtree is copied in, an INSERT is made on the node. Then, after completion of the INSERT action routine, ALOE looks for the next meta node in the inserted subtree causing ENTRY calls to the nodes within (but not at the root of) the inserted subtree.

5.2.6. FAILUP

An action call on FAILUP is made on an illegal **cursor-out** or **cursor-home** on a root of a window. **Cursor-next** and **cursor-previous** have no effect when the cursor is at the root of a window but are not considered illegal. More on window manipulation in chapter 9.

5.2.7. FAILDOWN

An action call on FAILDOWN is made on an illegal **cursor-in** on a terminal node or on a non-terminal node with no visible offspring (as specified in the current unparsing scheme, as described in section 3.1.4).

5.2.8. EDIT

An action call on EDIT is made after the .EDIT command. It indicates that the value of a constant has changed.

5.2.9. NEST

An action call on NEST is made after the .NEST command. The NEST call should behave as an implicit EXIT on the node being nested (the only non-meta node offspring of the resulting node or the node on which NEST is called), and an implicit ENTRY on the node being created by NEST.

5.2.10. TRANSFORM

An action call on TRANSFORM is made after the .TRANSFORM command. It should behave as an implicit ENTRY on the new node.

5.2.11. TDELETE

An action call on TDELETE is made before the actual transformation, but after it is determined that the transformation will succeed, *i.e.* all offspring of the node can be placed as offspring in the new node. It should behave as an implicit EXIT from the node.

5.3. Returns From Action Routines

Action routines can return four different types of values. Each return type is described below.

5.3.1. Abort

Applies to the CREATE, DELETE, and TDELETE calls. If the return value is minus two (-2) the operation should be aborted. In the CREATE case it means that the operation should be "undone" and a meta node should be placed back in that position. In the DELETE and TDELETE cases it means that the operation should not take place.

5.3.2. Go Ahead

If the return value is minus one (-1), ALOE should continue as if no errors had occurred, but still report errors if any occurred.

5.3.3. NIL

If the return value is NIL (zero), then if there were errors, the current node is set to the last error reported. If there were no errors, ALOE will continue its operation normally. ALOE behaves identically in this and previous case if there are no errors.

5.3.4. Address of Tnode

If the return value is greater than zero, it is interpreted to be a node address where the current node should be set to after reporting the errors, if any. This mechanism can be used to "abort" an EXIT or an ENTRY by returning the original node.

5.3.5. Error Reporting

In any of the above cases, if there were errors, ALOE will show them to the user one by one. After every error is displayed the user will decide if he wants to see the next error or to leave the error reporting. The choice will not be offered after the last error if the return value from the action routine and the node of the error are the same. See section 7.2 for an explanation of the different types of errors and formats.

5.4. Responsibilities of Action Routines

Each action routine has a variety of responsibilities, all of which are aimed at keeping the tree (and the user's view of the tree) consistent. Action routines are not supposed to delete the node they are called with. If this is desired on a CREATE call, the action routine should abort (*i.e.*, return a minus two).

5.4.1. Use of ALOELIB

All manipulations of the tree should be done through calls to routines supplied by the ALOE Library (see chapter 7). These are guaranteed to keep the syntactic consistency of the tree.

5.4.2. Calling Changedtree

If visible changes to the tree are made a call to the library routine `changedtree` (see section 7.9) must be made so that the display is correctly updated.

6. Extended Command Interface

An implementor of an ALOE may, in addition to attaching action routines to the operators, add extended commands. These extended commands may be used to manipulate the tree in any (preferably syntactically legal) way. The syntactic integrity is guaranteed if the implementation of these commands manipulate the tree using only the routines provided in the ALOE library (see chapter 7).

6.1. How to Add Extended Commands

Extended commands are easy to add to an ALOE. There is an extended command table that is linked into every ALOE. The standard extended command table is empty but new commands are added by creating a new one and linking it in before the standard archive (see chapter 11). The basic command table, **edcomtable**, and the extended command table, **extcomtable**, are arrays of type **struct edcommands**:

```
struct edcommands{
    char *comname;                    /* name of command              */
    char *syname;                     /* name of synonym              */
    struct tnode *(*comimpl)();       /* pointer to implementation    */
    int legal;                        /* true if execution currently ok */
    };
```

As shown, each command entry has four fields: First, the name of the command (comname) is given. This should be in capitals since ALOE folds all typed commands to upper case. Second, the synonym for the extended command (syname) is given (a null string indicates there is no synonym). The input handler of every ALOE understands that a ↑A (control-A, octal 001) is the first character of a two character extended command synonym. This means that it takes only two characters to specify a synonym starting with ↑A. If the full name is used then a user must type a "." in front of the command name (as is done for the standard editing commands to differentiate them from the constructive commands, see section 2.4). Third, the name of the procedure to be called (comimpl) to implement the extended command is given. And fourth, an integer (legal) indicating whether the command should be initially permitted or restricted is given. A non-zero entry indicates that it should be initially permitted. (For more information on how to permit and restrict editing and constructive commands, see section 7.1). Note that the final entry must contain all zeroes to mark the end of the list. There are no restrictions on the names of the commands except that if any one has the same name as a standard editing command, it can only be invoked using the synonym. For the names and synonyms of standard editing commands see section 2.4.

PRECEDING PAGE BLANK-NOT FILMED

A sample extended command table is:

```
struct edcommands extcomtable[]
        {"RESERVE","\001\022",exreserve,1,        /* ↑a↑r        */
         "DEPOSIT","\001\004",exdeposit,1,        /* ↑a↑d        */
         "RELEASE","\001\014",exrelease,1,        /* ↑a↑l        */
         0,0,0,0
         };
```

6.2. The Calling Sequence

Each extended command is passed the address of the node where it was invoked. Extended commands return a pointer to the new current node. It is perfectly reasonable to return the same node it was passed to it when no redirection is intended.

6.3. Responsibilities of Extended Commands

Each extended command has a variety of responsibilities, all of which are aimed at keeping the tree (and the user's view of the tree) consistent.

6.3.1. Use of ALOELIB

All manipulations of the tree should be done through calls to routines supplied by the ALOE Library (see chapter 7). These are guaranteed to keep the syntactic consistency of the tree.

6.3.2. Calling of Action Routines

Each extended command must interface with the action routines when it modifies the tree or when it returns a different node than the one it is passed (see chapter 5 for an explanation of the action routines interface). The routine `callonpath` from the library (see section 7.9) will make the appropriate EXIT and ENTRY action routine calls.

6.3.3. Calling Changedtree

If visible changes to the tree are made a call to the library routine `changedtree` (see section 7.9) must be made so that the display is correctly updated.

7. ALOELIB Routines

The ALOE library, ALOELIB, supplies a variety of routines needed to manipulate the tree constructed and edited by ALOE. They are provided for action routines and extended commands. In order to keep the tree syntactically correct at all times, it is essential to manipulate the tree using only the routines supplied by ALOELIB. ALOELIB provides an operational definition of the internal structure of ALOE.

The remainder of this section is broken up into logical groups of routines. The specification for the routines is given using the GC [Feiler 79] format with the types of the parameters specified in the parameter list. When a routine is said to return NIL (or `false`) it means that it returns zero. When it is said to return `true` it means that it returns a non-zero value.

7.1. Access Control Routines

This section describes the routines that are provided to control the commands and construction capabilities of an ALOE. The basic idea is that there is a stack of bit vectors that represent which commands are legal at various levels of the tree. The top of the stack indicates the current set of legal commands. The elements of this stack are of type `struct LegalAction *` (pointer to `LegalAction` structure, the details of the type structure are hidden). These elements should be created (by `topAction` or `newAction`) and pushed onto the stack whenever rights are changed (usually on an ENTRY action call) and popped off the stack when they are to be restored (usually on an EXIT action call). See chapter 5 for a description of the action routine calls.

Remember that DELETE usually acts as an EXIT and CREATE usually acts as an ENTRY. In all cases, be very careful if a command is aborted since this could cause the stack's integrity to be violated.

Initialize Access Stack

`initAction()`

Initializes the action stack. Must be called before the first use of the stack. This can be done on the CREATE action call at the system root.

Create New Access Element

`struct LegalAction *newAction()`

Returns a `LegalAction` that permits all operations to take place.

Restrict Command

`restrict(char *cmd; struct LegalAction *curAction)`

Cause `curAction` to be marked to not permit execution of `cmd`. The `cmd` must be an exact match of the command name in upper case in either the `edcomtable` or `extcomtable` arrays (see section 6.1).

Permit Command

`permit(char *cmd; struct LegalAction *curAction)`
Cause `curAction` to be marked to permit execution of `cmd`. Same restrictions as in `restrict`.

Restrict Construction

`consNOTOK(struct LegalAction *curAction)`
Cause `curAction` to be marked to not permit constructive commands.

Permit Construction

`consOK(struct LegalAction *curAction)`
Cause `curAction` to be marked to permit constructive commands.

Push Access Element

`pushAction(struct LegalAction *curAction)`
Push `curAction` onto the Action stack and set ALOE restrictions as listed in `curAction`.

Pop Access Element

`popAction()`
Pop the top element of the Action stack, delete it, and restore ALOE restrictions to the new top of the stack.

Take Top Access Element

`struct LegalAction *topAction()`
Return the current top of the Action stack. Used if you want to modify the current status rather than setting it absolutely.

7.2. Error Routines

This section describes routines that help the writer of action routines with interfacing to the error mechanisms provided by ALOE. Errors, warnings, and plain messages are queued in a buffer. Upon return from an action routine or an extended command, ALOE will display all the messages in order. (For more on the error reporting interface see section 2.12).

System Error

`syserror(char *msg)`
It should be called on finding an error in the internal tree structure or some other internal data structure. Prints out a header indicating that an internal system error has occurred followed by the string passed as a parameter. The program is not aborted.

Abort the System

`sysabort()`

Restores the terminal to a normal state and aborts the program.

Queue an Error

`error(char *msg; struct tnode *enode)`

An error message with text `msg` at node `enode` is queued for printing by ALOE after execution of the action routine or extended command. If the maximum number of errors permitted by the system is exceeded then the error is ignored and a "maximum number of errors exceeded" error is automatically added.

Queue a Parameterized Error

`error1(char *msg; struct tnode *enode; char *name)`

An error message is queued for printing by ALOE after execution of the action routine or extended command. The message is formatted by *sprintf*(a UNIX library routine) with `msg` as the formatting string and `name` as the single parameter after the formatting string. Exceeding the maximum number of errors is dealt with as in `error`.

Queue a Warning

`warn(char *msg; struct tnode *wnode)`

A warning with text `msg` at node `wnode` is queued for printing by ALOE after execution of the action routine or extended command. Exceeding the maximum number of errors is dealt with as in `error`.

Queue a Parameterized Warning

`warn1(char *msg; struct tnode *wnode; char *name)`

A warning message is given as described by `error1`. Exceeding the maximum number of errors is dealt with as in `error`.

Queue a Message

`message(char *msg; struct tnode *mnode)`

A message with text `msg` at node `mnode` is queued for printing by ALOE after execution of the action routine. Exceeding the maximum number of errors is dealt with as in `error`.

Queue a Parameterized Message

`mess1(char *msg; struct tnode *mnode; char *name)`

A message is given as described in `error1`. Exceeding the maximum number of errors is dealt with as in `error`.

Error Count

`int errcnt()`

Returns the number of errors, warnings, and messages queued so far.

7.3. List Manipulation Routines

This section describes routines that are available for the manipulation of variable arity nodes (also referred to as list nodes) in ALOE. See section 4.2.2 for a description of the internal representation of lists. All these routines report an error "node is not a list" and return NIL whenever the pointer passed to it is not a list (or a member of a list when the parameter is supposed to be one).

Add Meta to Front of List

`struct tnode *addfirst(struct tnode *list)`

Adds a new meta node to front of `list` and makes `list` its parent. Returns a pointer to the new meta node.

Add Meta to End of List

`struct tnode *addlast(struct tnode *list)`

Adds a new meta node to end of `list` and makes `list` its parent. Returns a pointer to the new meta node.

Get List

`struct listnode *getlist(struct tnode *list)`

Returns a pointer to first list header.

Check If List

`int islist(struct tnode *list)`

Returns `true` if `list` is a list and `false` otherwise.

Get List Length

`int lengthlist(struct tnode *list)`

Returns number of elements in `list`. Returns zero if `list` is empty (or it is not a list).

Find Element Index

int listindex(struct tnode *node)

`Node` is a member of some list. Returns its index in this list where i=0 for the first element. If `node` not found in the parent list reports "node not found in list" error and returns -1.

Apply Function to List Elements

`listwalk(struct tnode *list; int (*proc) ())`

Applies `int` function `proc` to every node in `list`.

Get Nth Element

`struct tnode *nthlist(struct tnode *list; int n)`

Returns the nth node in `list`, where n=0 is the first node. If `list` has fewer than `n` elements, reports "not found in list" error and returns NIL.

Get Nth Header

`struct listnode *nth(struct listnode *cell;int n)`

Returns the nth header after header `cell` where n=0 is `cell` itself. If there are less that `n` elements in the list, reports "not found in list" error and returns NIL.

Get Header of Element

`struct listnode *searchlist(struct tnode *node)`

`Node` is in some list. Finds and returns its header. If `node` is not in a list, reports "not found in list" error and returns NIL.

Get Next Header

`struct listnode *cdr(struct listnode *cell)`

Returns the next list header after `cell`. Returns NIL if `cell` is the last header of the list.

7.4. Filenode routines

This section describes the routines needed to manipulate filenodes (a filenode is often referred to as a `tnodef`). Filenodes are used to "partition" the internal tree into a database of separate files. The routines listed here provide all the necessary manipulation of file databases. In this section, *context* is the smallest subtree that the current node is in, where the root of the subtree is a FNONTERMINAL (a non-terminal node with a filenode associated with it, see section 3.2.3.6). A context stack is automatically kept by an ALOE. Elements are pushed on to the stack as contexts are entered and popped off when they are exited.

Check for Filenode

`int istnodef(struct tnode *node)`

Returns `true` if node is a filenode.

Get Current Context

`struct tnodef *curcontext()`

Returns the filenode of the current context.

Dirty Current Context

`cntxdirty()`

Sets the dirty bit of the current context. Will cause the writing of the subtree when it is exited.

Get Parent

`struct tnode *getfather(struct tnode *thisnode)`

Returns a pointer to the parent of `thisnode` skipping a `tnodef` if one exists. This procedure does not update either the context or symbol table stacks (see section 8.1). No files are written.

To Parent

`struct tnode *tofather(struct tnode *thisnode)`

Similar to `getfather`, but writes out both the current symbol table and tree if they have been changed. Both the symbol table and context stacks are popped. Most likely called by cursor movement or when the it is desired to get the effect of an implicit cursor movement.

Get Offspring

`struct tnode *getson(struct tnode *thisnode; int wson)`

Returns the `wson` offspring of parent node, where `wson`=0 for first offspring. Reports "son not found" error if parent has fewer than `wson` offspring and returns NIL. If the offspring is a `tnodef` then, if necessary, loads the subtree and returns its root. While the symbol table is loaded also, the symbol table stack is not pushed. The context is not changed.

To Offspring

`struct tnode *toson(struct tnode *thisnode; int wson)`

Similar to `getson`, but updates the context and symbol table. The old context is written if the tree or symbol table had been changed.

Getcar

`struct tnode *getcar(struct listnode *header)`

Equivalent to `getson`, but called from the list `header`.

Tocar

`struct tnode *tocar(struct listnode *header)`

Equivalent to `toson`, but called from the list `header`.

Get Parent of FNONTERMINAL

`struct tnodef *gettnodef(struct tnode *thisnode)`

Returns the filenode of an FNONTERMINAL. Reports "node not a FNONTERMINAL" error if `thisnode` is not and returns NIL.

Get Offspring of Filenode

`struct tnode *getfson(struct tnodef *thisnode)`

Returns the offspring (an FNONTERMINAL) pointed to by a `tnodef`.

Find Filenode Above Node

`struct tnodef *findfnode(struct tnode *node)`

Finds the `tnodef` which is the context for `node`. If the `node` is a filenode, the filenode which contains it is returned.

Check Point Subtree

`CheckPoint(struct tnode *node)`

Writes the file which contains `node`. The `node` may be a filenode in which case it writes the file of the context that contains the filenode and not the one of the subtree associated with the filenode.

7.5. Node Management Routines

This section describes routines that are used to manipulate nodes and subtrees of the tree.

Check for Meta

`int ismeta(struct tnode *node)`

Returns `true` if `node` is meta and `false` otherwise.

Find Meta

`struct tnode *findmeta(struct tnode *node)`

Performs preorder search for first meta node in subtree whose root is `node`. If meta node is found, returns pointer to it; if none found, returns NIL.

Make a Node

`struct tnode *chkmake(int opt; struct *thisnode; char *valuestr)`

Validates the creation of operator `opt` at `thisnode` and creates the node if it is legal. `Thisnode` must be a meta node. If `valuestr` is NIL then values will be read from input if they are needed. Returns pointer to new node, unless an error occurs in which case it returns NIL. `Chkmake` does *not* call action routines.

Copy a Subtree

`struct tnode *chkcopy(struct tnode *source, *dest; int doact)`

Copies the subtree from `source` to `dest`. `Dest` must be a meta node. If `doact` is `true`, it calls action routines on INSERT as the nodes are created. Returns NIL if an error occurs, otherwise it returns a pointer to the newly inserted subtree.

Delete a Subtree

`struct tnode *delsubstree(struct tnode *thisnode)`

Deletes the subtree pointed to by `thisnode` and replaces it with the proper meta node. Releases subtree space. Returns pointer to the new meta node. If thisnode is a meta node and a member of a list, then it deletes the node, does not replace it with a meta, and returns a pointer to the left sibling if one exists or to the parent otherwise.

Automatic Application of Operators

```
struct tnode *applyauto(struct tnode *thisnode;int doact)
```

Traverses the subtree starting at `thisnode` and looks for meta nodes that can be applied automatically: They can be replaced by nonterminals in a class where there is only one choice (more on automatic application in section 2.3). Does not apply terminals, FNONTERMINALS, or "non-visible" nodes (see section 3.1.4).

Apply Function to Each Element of Subtree

```
TreeWalk(struct tnode *thisnode; int (*proc)())
```

Walks the tree starting at `thisnode` in preorder calling the procedure `proc` at every node passing the node as parameter.

Remove All Metas in List

```
RemAllMetas(struct tnode *list)
```

Removes every element of `list` that is a meta node.

Almost Remove All Metas in List

```
RemoveMetas(struct tnode *list)
```

Removes every element of `list` that is a meta node unless this would leave `list` empty. In that case, it leaves one meta node in `list`.

Get Type of Node

```
char terminal(int optype)
```

Returns value of `terminal` field from operators table for operator `optype`. The possible different values for this field are defined in the file "*/usr/gandalf/include/ALOELIB.h*".

Get Arity of Node

```
int arity(int optype)
```

Returns value of `arity` field from operators table for operator `optype`. The values for arity are: zero if *terminal* or *variable arity* non-terminal, and the number of offspring for *fixed-arity* non-terminals.

Get Operator Name

```
char *opname(int optype)
```

Returns a pointer to the name of the operator `optype` from table of operators.

Get System Root

```
struct tnode *getsysroot()
```

This returns the current value for the root of the entire system.

Get Window Root

`struct tnode *getroot()`

This returns the root of the current window.

7.6. Space management routines

This section describes routines involved with the space management of strings. All other space management is dealt with automatically by `chkmake`, `chkcopy` and `delsubtree`.

Free String

`freestring(char *it)`

Frees the space allocated to a string.

Create String

`char *newstring(char *oldchar)`

Allocates space for a new string. Copies string into allocated space.

7.7. Status

This section describes the routines provided for manipulation of the status fields of tree nodes. They may be used by an ALOE implementor in cases where the status checking needed for trees is relatively simple. In other cases, the implementor should write and use more complex status schemes.

All the routines in this section access the status field by using the `tnstat` type definition (see section 4.5). The two fields manipulated are `actstat` and `count`. The `actstat` field can contain three values: OK, NOTOK, and UNK. This indicates the current status of the `tnode` (UNK indicates the status is currently unknown). The `count` field is the number of offspring of the node that have a status of OK. The routines described below manipulate these fields with these semantics in mind.

Initialize Status

`initstat(struct tnode *node)`

Initializes the status of `node` by setting `actstat` to NOTOK and `count` to zero. Also, if `node` is an element of a list, `statusNOTOK` is called on its parent because the length of the list has now changed. In general, `initstat` is called on a node when it is created (*i.e.*, a call to `initstat` should be made in the CREATE case of the appropriate action routines).

Set Status OK

`statusOK(struct tnode *node)`

This routine increments the `count` of the parent of `node` and sets the `actstat` field of `node` to OK.

Set Status Not OK

`statusNOTOK(struct tnode *node)`

This routine makes sure that `node` and all of its ancestors are set to NOTOK. This is done by chasing up the tree setting the ancestors to NOTOK until one that is already set to NOTOK is found. Each node which is set to NOTOK also has its parent's `count` decremented. This routine should be called in the DELETE case of appropriate action routines.

Check for Status?

`int checkit(struct tnode *node; int nsons)`

This routine returns `true` (non-zero) or `false` (zero) depending on whether or not the `node` should be checked for valid status yet. If it should be checked for status, it is up to the calling action routine to do so and call `statusOK` is the check is successful. The `nsons` parameter is -1 in the case where `node` is a list. In this case, `checkit` will return `true` (indicating the list should be checked for validity) if `actstat` of `node` is NOTOK (if it is OK then there is no reason to check it again) and if the `count` field of `node` is equal to the length of the list (which indicates that every list element has a valid status and thus the list might be valid). If `nsons` is greater than zero, then `checkit` returns `true` if `actstat` is NOTOK and `count` is equal to the value of `nsons`. This routine is usually called in the EXIT case of appropriate action routines. The code in this case should look something like:

```
case EXIT:
    if (!checkit(node,nsons))
        break;
    if (/* the status of the tree is valid */)
        statusOK(node);
    break;
```

7.8. Window Manipulation Routines

This section describes the routines needed to manipulate the windows on the screen. The window manipulation and screen layout are discussed in detail in chapter 9.

Look Up Window

`struct window *LkupWindow(char *windname)`

Returns a pointer to the window structure with name `windname`. Returns NIL if window is not found.

Assign Text Window

`AsgTextWindow(struct window *wind)`

Marks window `wind` to be a text window.

Assign Tree Window

`AsgTreeWindow(struct window *wind; int scheme; struct tnode *rnode; struct tnode *cnode)`

Marks window `wind` to be a tree window. Assigns the values of `scheme` to the unparsing scheme associated with the window, `rnode` to the root node of the window, and `cnode` to the current node of the window.

New Tree Window

`NewTWindow(struct tnode *rootnode; int scheme; char *wname)`

Allocates a new window from the pool of tree windows with name `wname`, marks it as tree window, and pushes it into the context window stack. Assigns the value of `scheme` to the unparsing scheme associated with the window. Assigns the value of `rootnode` to both the root node and the current node of the window.

Release Tree Window

`struct tnode *RemTWindow()`

Pops the context window stack. Returns the current node of the window on the top of the stack (after the pop).

Set Current Window

`struct window *SetWindow(struct window *wind)`

Makes window `wind` to be the current window. Returns a pointer to previous current window.

Force a Tree Window

`ForceProg()`

Makes the top of the context window stack to be the current window regardless of the value of current window.

Release Window

`RelWindow(struct window *wind)`

Marks the window `wind` to be "free".

Set Output Window

`int setoutwind(char *wname; integer clear)`

Sets the current *user* window to `wname`. `Wname` must be defined in the window definition file (see chapter 9). `Clear` indicates if the window should be cleared after setting or if the cursor should be put back where it was last time `resetoutwind` was called. Output window should be reset using `resetoutwind` before control is passed back to ALOE.

Returns NIL in case of error. Output to the window can then be sent as if it was sent to a terminal (using *printf* for example). This routine depends, of course, on the hardware capabilities of the terminal. The current implementation relies on the Concept 100 capabilities. See chapter 10.

Reset Output Window

`resetoutwind()`

Resets the current user window set by `setoutwind`. Calls to `setoutwind` and `resetoutwind` should be done in pairs since no explicit stack is kept.

7.9. Miscellaneous Routines

Indicate Tree Has Changed

`changedtree()`

This is used to indicate that an action routine or an extended command has modified the tree is some way and that the changes should be displayed.

Call Action Routines on Path

`struct tnode *callonpath(struct tnode *fromnode, *tnode)`

Makes the action routine calls on EXIT and ENTRY on the nodes of the tree path from `fromnode` to `tonode`. Reports all the errors encountered in the process and returns the node of the last error reported or `tonode` if no errors were found. For a detailed description of the action routines interface see section 5.

Display a Tree

`printree(struct tnode *rootnode; struct tnode *hnode)`

Unparses a subtree with root `rootnode` highlighting `hnode` (to be displayed in the current window). This routine is normally called only by `PrintError` (see section 8.5), after setting the current window to the *error* window (see section 9.1).

Set Data Base Name

`setdb(char *name)`

Sets the current system name to `name`.

Get Full Pathname

`fullpath(char *final, *relative)`

Concatenates the current system name with the `relative` path and copies it into `final`. File paths in filenodes are kept relative to the system name.

8. Environment Specific Routines

Experience has shown that some ALOEs desire effects that can not be produced simply through calls to action routines and extended commands. Therefore, the system provides a variety of routines which the ALOE implementor can replace. In order to relieve the implementor of a simple ALOE from writing these routines every time, a set of standard routines is provided. This chapter describes which routines are provided, how they are intended to be used, and what the standard routine that is provided does.

When a particular ALOE is going to be generated, some or all of the routines can be replaced with environment specific implementations. The object files for the standard supplied routines are collected in an archive in *"/usr/gandalf/lib/DEFAULT.a"*. The object files for environment specific versions of these routines must be given to the loader before DEFAULT.a. The loader will not pick routines from DEFAULT.a if they have already been defined. See appendix I for an example of some simple changes to the symbol table manipulation routines. See [Notkin 82] for a detailed description of complex implementations of environment specific routines in an ALOE.

8.1. Symbol Table

When terminals that are variables are used in the grammar definition, this implies that symbol table actions be taken to help manipulate these nodes. This section describes the standard actions that are taken and the structure definitions on which these actions take place. Although the provided routines are often trivial in the standard case, the mechanisms are provided so that far more complicated symbol tables may be created and manipulated by an ALOE.

8.1.1. Symbol Table Definitions

This is the standard definition for symbol tables and symbol table entries. They are the definitions used for stand alone ALOEs. Environment specific symbol tables can be developed by extending these definitions (with different names). Routines have to be written to manipulate the environment specific symbol tables. (See appendix I for a simple example of such an extension.)

The first definition, `symboltable`, is quite simple. It contains a table type indicator (there is only one table type in the standard implementation), a bit indicating whether or not the symbol table has been changed since it has last been written out, and a pointer to a linked list of symbol table entries.

```
struct symboltable{
    char tabletype;                    /* type of symbol table      */
    char dirtybit;                     /* modification since write? */
    struct symtabentry *entries;       /* pointer to first entry    */
    };
```

The second definition, symtabentry, describes the actual references within the symbol table. The first field points to the print name of the symbol. This is used for unparsing, searching, *etc.* The second field is the link field to the next symtabentry in the symbol table. The third field is a back pointer to the symboltable entry which contains this entry. The fourth entry points to the tnodev (see section 4.3.3) which defined the variable described by this symbol table entry. And the final field is a reference count that indicates how many instances of the variable are currently in the tree.

```
struct symtabentry{
    char *myname;                    /* print name of symbol         */
    struct symtabentry *nextsymbol;  /* pointer to next entry        */
    struct symboltable *mytable;     /* pointer to symbol table      */
    struct tnodev *mynode;           /* pointer to symbol . ''nition */
    int refcount;                    /* reference count              */
    };
```

8.1.2. Symbol Table Manipulation Routines

The routines that are called to manipulate the symbol tables and their entries are described next. For each routine, a description of that it does in the standard case and when it is called in all cases is given.

Create a Symbol Table

`struct symboltable *createtable(struct tnodef *node)`

This routine is called when a new filenode (tnodef) is created. Hence, it can be used to create a multi-level symbol table based on particular contexts defined by levels of filenodes. Node is supplied so that a reference to the new symbol table can be stored in the provided field of the filenode structure definition (see section 4.4). The standard case takes no action.

Delete a Symbol Table

`deltable(struct symboltable *table; char *filename; int sontype)`

This routine is called whenever a filenc le is deleted. The first parameter, table, is a pointer to the table to be deleted. This is taken from the filenode that is being deleted. The second parameter indicates that name of the file that contains the symbol table to be deleted. This name is the full path name to the file referenced in the filenode with ".st" appended to it. The third parameter, the sontype, is taken from the stype field of the filenode being deleted. This routine can therefore be used to delete the symboltable associated with a filenode. The standard case takes no action.

Create a Symbol Table Entry

`struct symtabentry *newstentry(struct symboltable *table;char *name)`

The first parameter indicates which symbol table to enter it in and the second parameter is the print name to give it. In the standard case, this is called by putname (defined below) if a new symbol is entered into the symbol table. The standard implementation also sets the reference count to one, links the entry back to the symbol table, and links it into the symtabentry list. The value of the new entry is returned so that the value of the defining tnodev may be set.

Free a Symbol Table Entry

`freestentry(struct symboltable *table; struct symtabentry *entry)`

The first parameter is the symbol table where the `entry` to be deleted appears and the second parameter is the `entry` itself. The standard implementation unlinks the `entry` from the list pointed to by `table` and frees the space of the `entry`. In the standard case, `freestentry` is called only if the reference count of the entry is zero after it is decremented by `remname`.

Put a Name

`struct symtabentry *putname(char *name; struct tnodev *vnode)`

This routine is called directly from ALOE when a variable is entered. Its intended use is to enter `name` into the symbol table. The standard case checks for existence of `name` in the current symbol table. If it is there, the reference count is incremented. Otherwise, a new entry is added to the symbol table through a call to `newstentry`.

Remove Name

`remname(struct symtabentry *entry;struct tnodev *node)`

This is called by ALOE when a variable is deleted and is intended to be used to delete a name from the symbol table. The standard case decrements the reference count of the `entry`. If it is zeroed by this decrement, then `freestentry` is called to delete the entry from the symbol table and to free the space used by the entry.

Get Name of Entry

`char *getname(struct symtabentry *entry)`

This is called by all routines which need to get the print name of a `tnodev`. The standard version returns a pointer to the print name of `entry`.

Store Symbol Table

`storetable(struct symboltable *table; char *filename; int sontype)`

This is called when the subtree represented by a filenode is to be written (*i.e.*, checkpointed). It is intended to write out symbol table `table` on file `filename`. The `table` and `sontype` parameters are taken from the equivalent fields in the filenode being written. The `filename` parameter is the full path name of the file name listed in the filenode with ".st" appended. Since symbol tables are not stored in the standard case, the standard implementation is an empty routine.

Read Symbol Table

`struct symboltable *loadtable(char *filename)`

This is called when the subtree represented by a filenode is read in. It is intended to be used to load the sorted symbol table that is associated with the subtree being read in. The `filename` is the name of the file that contains the symbol table to be read in. The name of the symbol table file is created as in `storetable`. The standard implementation is an empty routine.

Store Name in Symbol Table File

`storename(struct symtabentry *entry; FILE *filebuf)`

Write symbol table entry into the symbol table storage file. This is provided so that representations other than the print name of the variable may be stored in the symbol table file. The standard implementation simply places the print name of the `entry` into the file indicated by `filebuf`.

Read Name from Symbol Table File

`struct symtabentry *loadname(struct tnodev *node; FILE *filebuf)`

Read symbol table entry from the symbol table storage file. This is provided so that alternate representations of variables stored by `storename` can be read. The standard implementation simply reads a name from the file and uses this as the print name of the variable.

Push Symbol Table

`pushtable(struct symboltable *table)`

This is intended to support a stack of symbol tables (along with `poptable`). It is called whenever the current symbol table may be changed. Whenever a new context is entered (*i.e.*, a filenode has been entered), the new symbol table (found in the filenode entry) is pushed.

Pop Symbol Table

`poptable(struct symboltable *table)`

This is the matching routine to `pushtable` described above.

Resolve Multiple Symbol Table References

`Resolveme(struct tnodef *thisnode)`

This is called whenever a filenode's subtree is read in. `Resolveme` can be used to resolve entries among interconnected symbol tables.

8.2. Filenode Support

Get Filenode Print Name

`char *getprname (int optype; struct tnode *themeta)`

Called by ALOE when an FNONTERMINAL (see section 4.4) is created by `chkmake` (see section 7.5) with a NIL `valuestr`. It is used to allow an ALOE to set names of files for filenodes so that conflicts among these files may be avoided. The first parameter indicates the `opt` field of the node being created. This permits different actions to be taken on creation of different types of nodes. The second parameter is the node that is going to be replaced. It should call `setdb` (see section 7.4) to set the current system name when called for the system root. In the standard case, `getprname` prompts "Program Name" and checks for duplication of this name in the current directory. Any ALOE that has filenodes other than at the root should replace this routine.

Faildown on a Filenode

`struct tnode* tffaildown(struct tnodef *thefnode)`

Called by ALOE when a **cursor-down** is attempted on a filenode whose FNONTERMINAL contains an "`@x`" in the current unparsing scheme. Systems that use multiple tree windows may change the unparsing scheme and return the offspring of the filenode after creating a new window (see section 9.2.1.2 for an example). In the standard case `tffaildown` reports that it cannot change the unparsing scheme.

8.3. Filepath Routines

Make System Path

`makepath(char *fname)`

Called by ALOE to construct the path to the tree file containing the description for a separately stored subtree of the system. In the standard case, this description is in file `fname` so the standard implementation returns `fname` itself. In more complex systems, the description may well be in directory `fname` so `makepath` may want to append a "`/`" onto the filename.

Current Path

`char *curpath(char *prname;int stype)`

This is called whenever an ALOE needs to access a UNIX file. It is intended to permit construction of complex UNIX pathnames that support mappings to the UNIX file system of files manipulated by an ALOE (*e.g.*, tree files and symbol table files). The `stype` is passed so that, depending on the context (*i.e.*, the kind of filenode) a different current path name can be built. This routine is responsible for creating any directories that are needed. Returns `prname` itself in the standard case.

Delete Path

`delpath(char *path; int stype)`

It is intended to allow deletion of directory structures when a filenode is deleted. In the standard case it does nothing.

8.4. Initialization Routines

Ginit

`Ginit()`

This routine is called by ALOE after doing its own initialization but before doing anything else. May be used to load some environment specific information or to initialize needed variables. The standard `Ginit` initializes the variable "`STANDALONE`" to thirty. Elaborate ALOEs will very likely be partitioned into different files using the filenode facility which handles checkpointing automatically. These ALOEs do not have to initialize "`STANDALONE`" unless they want checkpointing of the top-level subtree. Setting the variable to a positive value sets the frequency of checkpointing to that value.

Gquit

`Gquit()`

Before leaving ALOE after a .QUIT command, ALOE calls this routine. It should be used to save whatever state needs to be saved and to clean up system specific information. The standard routine does nothing.

Gparse

`int Gparse(char *textfile)`

When ALOE is called with a -p switch (see section 2.15.2) it passes the text file to this routine for parsing. `Gparse` is responsible for building an internal structure for the text form of the tree. In case of an error in `Gparse`, an ALOE will call the UNIX system library routine exit(1). The standard *Gparse* returns an error since it means that the parse flag was used without a parser being supplied. Details on how to implement and interface a parser are available from the authors.

8.5. Miscellaneous Routines

Print Errors

`PrintError(struct tnode *Root; struct tnode *node)`

`PrintError` is called by the error reporting mechanism of an ALOE. It is provided so that ALOEs with a complex window structure may display the error based on a *root* (some predecessor of `node`) that will show the appropriate context. In the standard case calls routine `printree` (see section 7.9) to unparse the program starting at `Root` and highlighting `node`.

Load the Window Definitions

`char *getWindowFile()`

Called on startup of an ALOE. Intended to load the window layout definitions. In the standard case, `getWindowFile` uses the environment variable ALOEWINDOWS to locate the file from which to load these definitions. If it is not defined, it loads from *"/usr/gandalf/.windows"*. An environment specific routine may use a different environment variable and a different default layout file.

9. Windows

This chapter describes those aspects of the windows that impact the generation of a particular ALOE. For a complete description of the window manipulation package of ALOE see [Feiler 81].

9.1. Display Layout

The display layout for ALOE is parameterized and may be redefined for every ALOE instantiation. On invocation, in the standard case ALOE reads the layout description from the file specified in the environment variable ALOEWINDOWS. If such variable is undefined then it picks it up from the file "*/usr/gandalf/.windows*" (see section 8.5).

The definition file has the following format: window name, coordinates of upper left hand corner, size in lines, and size in columns, all enclosed in curly brackets. The last line in the file must be "`%end%`". The windows form a pool from which tree windows are allocated (see section 9.2 below). The *clipped* windows form a pool from which clipped area windows are allocated. They are also tree windows but they are normally displayed on a different section of the screen. Action routines have no control of clipped windows. No action routine calls are made while in a clipped window (see section 5.1).

The *root* window is the first tree window (used for the system root). The *help* window is used to display "help" information by the commands .HELP and .? and is normally at the right end of the screen (see section 2.4). The *command* window is used for command input. It must be two lines long. It normally is at the bottom of the screen. The *status* window is a one line window that displays status information (see section 2.13). The *windows* window (also referred to as the *context* window) is used in connection with the *show-windows* mode (see section 2.11) to display the context of tree windows.

The *user* window is an optional window used for input/output of user programs that are interpreted or run from an ALOE environment. (See appendix I for an example of its use). Its name does not necessarily have to be *user*. The *error* window is a tree window used for displaying errors (see section 2.12). Routines to manipulate all these windows are provided as part of the ALOE Library (see section 7.8).

This is the default layout (from "*/usr/gandalf/.windows*"):

```
{ error , 1 , 1 , 20 , 79 } ,
{ command , 23 , 1 , 2 , 79 } ,
{ status ,  22 , 1 , 1 , 79 } ,
{ windows ,  21 , 1 , 1 , 79 } ,
{ help   ,  1 ,  61 , 21 , 19 } ,
{ user   , 1 , 61 , 20 , 18 } ,
{ root , 1 , 1 , 21 , 79 } ,
{ eMpty , 1 , 1 , 20 , 79 } ,
{ eMpty , 1 , 1 , 20 , 79 } ,
{ eMpty , 1 , 1 , 20 , 79 } ,
{ eMpty , 1 , 1 , 20 , 79 } ,
{ eMpty , 1 , 1 , 20 , 79 } ,
{ eMpty , 1 , 1 , 20 , 79 } ,
{ eMpty , 1 , 1 , 20 , 79 } ,
{ eMpty , 1 , 1 , 20 , 79 } ,
{ clipped , 12 , 1 , 10 , 59 } ,
{ clipped , 12 , 1 , 10 , 59 } ,
{ clipped , 12 , 1 , 10 , 59 } ,
{ clipped , 12 , 1 , 10 , 59 } ,
{ clipped , 12 , 1 , 10 , 59 } ,
{ clipped , 12 , 1 , 10 , 59 } ,
{ clipped , 12 , 1 , 10 , 59 } ,
{ clipped , 12 , 1 , 10 , 59 } ,
{ clipped , 12 , 1 , 10 , 59 } ,
{ clipped , 12 , 1 , 10 , 59 }
%end%
```

9.2. Tree Windows

Often the tree that is manipulated by an ALOE is too large to be displayed on the screen. Since, in addition, there are often logical places where the tree can be broken into levels, routines are provided that permit the implementor to manipulate a stack of windows (also referred to as context window stack) that show different levels of the tree in different windows (most often overlayed). The level of detail shown is controlled by the unparsing schemes. Action routines are invoked when the top or bottom of a given level of detail has been reached and they in turn can invoke the routines to change windows in the screen (see section 7.8). This section describes how to create and manipulate these windows from the appropriate action routines.

9.2.1. Pushing Windows

Windows are usually pushed onto the stack when an attempt is made by the user to see something that is not currently displayed on the screen. This is due to the fact that the current unparsing scheme is displaying a node that displays a constant string instead of displaying its offspring. When a **cursor-in** is tried at such a point either a FAILDOWN action call or a call to `tffaildown` (if the current node is a filenode) is made. This call can then create a new window and then set this new window to be the current window. The two situations are as follows:

9.2.1.1. FAILDOWN

When the current node is a regular nonterminal, the FAILDOWN action call is used to create a new window. The code will look something like:

```
case FAILDOWN:
        NewTWindow(thisnode,1,"WindowName");
        break;
```

`NewTWindow` creates a new window with the first parameter as the root, the second parameter as the new unparsing scheme for the new window, and the third parameter as the window name (see section 7.8). Window names should be unique in order to permit access by the window changing commands.

9.2.1.2. Filenode Faildown -- tffaildown

On the other hand, if the node where the **cursor-in** is attempted is a filenode, no action call is made. Instead, the routine `tffaildown` is called. This routine should look something as follows (a standard implementation that does nothing is provided, see section 8.2).

```
struct tnode *tffaildown(struct tnodef *thisnode)
{
    struct tnode *RootNode;
    char *WindowName;

    RootNode = getfson(thisnode);
    WindowName = opname(thisnode->stype);
    NewTWindow(Rootnode,1,WindowName);
    return RootNode;
}
```

This version of `tffaildown` says that every filenode in the grammar associates a new window with the subtree the file represents. If only some of the filenodes are to be placed in new windows, then the `stype` of the filenode can be checked to see whether or not `NewTWindow` should be called. This routine must be rewritten if filenodes are recursively nested since there would otherwise be a naming problem for the windows.

9.2.2. Popping Windows

When the user is done with a window (indicated either by a **cursor-up** or **cursor-home** from the window root or else by a .WINDOW command), the window must be popped from the stack. Here, an action routine call on FAILUP is made. Again, two slightly different structures are needed to deal with the problem of filenodes.

9.2.2.1. Regular Nodes

In the standard case, the simple call

```
case FAILUP:
    RemTWindow();
    break;
```

will be sufficient to pop the current window off the stack and return to the previous window. This is only true if the root of the current window is the same node as the leaf of the previous window where the new window was created. If this is not the case, then follow the directions given in the filenode case.

9.2.2.2. Filenodes

Filenodes are slightly different because the actual node is the root of the current window and the filenode associated with the node is the leaf of the previous window. Hence, the popping of the window alone is insufficient since ALOE would still think the current node was set to the actual node (which is no longer on the screen). Hence, the sequence

```
case FAILUP:
    return(RemTWindow());
```

will pop the window and set the new node to the old leaf (*i.e.*, the filenode) in the previous window. This is due to the fact that the return value of an action routine, if greater than zero, is the address of the new current node (see section 5.3).

10. Display Handling

The display in ALOE is updated after every interaction using a display package written by James Gosling [Gosling 81b]. The package uses an optimal algorithm for redisplaying the screen every time it changes. The package also implements the highlighting of the area cursor of ALOE. Peter Feiler augmented the package to deal with the different windows defined in ALOE and with its layout organization.

The display package includes definitions for several terminals that can be supported. The implementation of the cursor highlighting, however, is specific for the C-100 family of terminals (using the *set attribute of block* function). If an ALOE is to be generated to run on a terminal other than the C-100 the display package has to be modified in two ways: The definition of the terminal has to be entered (chances are it is already there) and the procedures that handle the highlighting have to be modified. This type of modification has already been done to interface to the Alto AT [Ball 80] display system.

More information on the necessary modifications of the display package for interfacing to other terminals is available from the authors.

11. How to Build an ALOE

11.1. What Commands are Available

The following commands are all available from the "*/usr/gandalf/bin*" directory.

11.1.1. Invoking the ALOE Generator

The first step in generating an instantiation of an ALOE is to create (or modify) a grammar using the ALOE generator. This is currently available as *aloegen* in "*/usr/gandalf/bin*". The tree file that *aloegen* saves will be named as the name given by the user to *aloegen* with extension ".tr".

11.1.2. Make Tables

After the grammar has been created using *aloegen*, the tables that ALOE needs to instantiate the language are generated by executing the command

```
maketables <language name>
```

where `<language name>` is the name of the tree file (without the extension) created by *aloegen*. This command will leave two files on the current working directory -- `<language name>`tbl.o (the compiled tables) and `<language name>`.infop (a set of definitions which indicate the values assigned to the operators of the language). The *infop* table is described in more detail below.

11.1.3. Load ALOE

In order to link an ALOE, the *ldaloe* command is used. The basic command is simply

```
ldaloe <language name> [<.o and .a files>]
```

which generates an ALOE in the file named `<language name>`.aloe. If the ALOE being built uses action routines, has extended commands, or includes any environment specific routines, these must also be provided to *ldaloe*. These can be in either .o format or .a (archive) format and simply need be listed after the `<language name>`.

11.1.4. Build an ALOE

Often it is desirable to combine the making of the tables and the loading of an ALOE. The *buildaloe* command provides this function. The command is:

```
buildaloe <language name> [<.o and .a files>]
```

This simply applies *maketables* and then applies *ldaloe*.

PRECEDING PAGE BLANK-NOT FILMED

11.2. Files to be Included

The following definition files (except for the *infop* file which is language dependent) are available in "*/usr/gandalf/include*".

11.2.1. The Infop File

The *maketables* command generates the file <language name>.infop. This is used by action routines and extended command routines in order to determine (from the `opt` field of nodes (see section 4.1)) which operator a particular node represents.

11.2.2. ALOELIB.h

The extern definitions for the routines supplied by ALOELIB (see chapter 7) are found in the file "*/usr/gandalf/include/cALOELIB.h*". The extern definitions from ALOELIB that are used for the language GC (Gandalf C) are in the file "*/usr/gandalf/include/ALOELIB.h*".

Also included in these include files are several sets of definitions frequently needed by action routines and extended command routines. The first set consists of the definitions that indicate the type of the action routine call (*e.g.*, ENTRY, DELETE). These are most commonly used in switch statements on entry to action routines. The second set of definitions indicate the type of each node as described in the operators table. These indicate whether a node is a STATIC, a META, *etc.* Finally, these include files provide all the need structure and type definitions needed to implement action routines and extended commands.

Acknowledgements

We would like to thank Nico Habermann, Bob Ellison, Peter Feiler, Gail Kaiser, David Garlan, Barbara Denny, and Steve Popovich for their valuable contributions in the development of and experience with ALOEs. We also greatly appreciate their comments on early drafts of this manual.

Several other contributions are especially noteworthy:

- Gail Kaiser built the initial version of the ALOE library.
- Bob Ellison implemented the filenode support routines of the ALOE library.
- Peter Feiler wrote the window package and its interface to the display package.
- Phil Wadler implemented the initial ALOE preprocessor which was the predecessor of the ALOE generator.

Appendix I
An Example of an ALOE

The following example is an instantiation of ALOE that implements a small interpreter. The program is not intended to be overly realistic but rather it is intended to be a model for people who are building their first ALOE.

I.1. Overview of the example

The language provides two statement types -- *print* and *for*. The *print* statements take a list of expressions to be evaluated and displayed. The *for* statements are of the common BASIC type with a loop variable, a termination value, an increment, and a list of statements to be executed. Variables are defined only by *for* statements. Other expression elements are integers and expressions formed by addition and multiplication operators.

The example has a simple grammar that shows both types of non-terminals (lists and fixed arity nodes) and three of the most common types of terminal nodes (static, constant, and variable). One routine from the standard environment specific symbol table routines is replaced, one extended command is provided, and one action routine is given. All the operators except *for* have one unparsing scheme. A second scheme for the *for* statement is given just to show how an alternate concrete form can be given.

The grammar is shown first and the code to implement the interpreter is given after.

I.2. The Example Grammar

```
Language Name: INTERP
Root Operator: PROGRAM

{       /* terminal operators */

LOOPVAR =     {v}
            | (0) "@s"
            | action: <none>
            | synonym: "'" ;
INT =         {c}
            | (0) "@c"
            | action: aINT
            | synonym: "#" ;
EMPTYSTEP =   {s}
            | (0) "1"
            | action: <none>
            | synonym: <none> ;
}
```

PRECEDING PAGE BLANK-NOT FILMED

```
{       /* non-terminal operators */

PROGRAM =      stmts
             | (0) "@1"
             | action: <none>
             | synonym: <none>
             | precedence: <none>
             | Filenode;
PRINT =        <exp>
             | (0) "print @0,"
             | action: <none>
             | synonym: <none>
             | precedence: <none>
             | Non-filenode;
FOR =          loopvar exp exp stepexp stmts
             | (0) "for @1 = @2 to @3 step @4@+@n@5@-"
               (1) "for (@1 = @2; %1 <= @3; %1 =+ @4)@+@n@5@-"
             | action: <none>
             | synonym: <none>
             | precedence: <none>
             | Non-filenode;
PLUS =         exp exp
             | (0) "@1 + @2"
             | action: <none>
             | synonym: "+"
             | precedence: 1
             | Non-filenode;
TIMES =        exp exp
             | (0) "@1 * @2"
             | action: <none>
             | synonym: "*"
             | precedence: 2
             | Non-filenode:
STMTS =        <stmt>
             | (0) "@0@n"
             | action: <none>
             | synonym: <none>
             | precedence: <none>
             | Non-filenode;
}

{       /* classes */
stmts          =       STMTS ;
exp            =       INT LOOPVAR PLUS TIMES ;
loopvar        =       LOOPVAR ;
stepexp        =       INT PLUS TIMES EMPTYSTEP ;
stmt           =       PRINT FOR ;
}
```

1.3. An Example Program

The two programs below have identical internal structures but have been unparsing according to different unparsing schemes. This is the program generated by unparsing scheme zero:

```
for i = 4 to 8 step 2
    print (i + 3) * i, i
```

The program generated by unparsing scheme one is:

```
for (i = 4; i <= 8; i =+ 2)
    print (i + 3) * i, i
```

In addition to the use of multiple unparsing schemes, note how the precedence values are used to provide the automatic generation of parentheses.

1.4. The Example Code

```
/* This include gets the needed structure definitions, extern
   definitions for ALOELIB routines, etc.  Details on the
   function calls can be found in chapter 7. */

#include "/usr/gandalf/include/cALOELIB.h"

/* This include gets the definitions of the opt fields that
   maketables created for each node type.  They are of the
   form iNAME where NAME is the name of the operator.  For
   instance, an opt field value of iFOR in a tnode indicates
   that the node is a FOR operator. */

#include "example.infop"

#define NIL 0

/* These are definitions which permit calls to getson to use symbolic
   names for the son number.  These only need be defined for fixed
   arity nodes since lists are accessed with getlist. */

#define PROGRAMstmts    0
#define FORloopvar      0
#define FORfromexp      1
#define FORtoexp        2
#define FORstepexp      3
#define FORstmts        4
#define PLUSexp1        0
#define PLUSexp2        1
#define TIMESexp1       0
#define TIMESexp2       1

/* These are definitions to make the casting of these types more readable */

#define Isymtab(entry) ((struct Isymtabentry *)entry)
#define Symtab(entry)  ((struct symtabentry *)entry)
#define Tnodev(node)   ((struct tnodev *)node)
#define Tnodec(node)   ((struct tnodec *)node)
```

```
/* The following type definition is simply an extension of the standard
   symbol table entry definition.  It now contains a value field in which
   integer values are stored. */

struct Isymtabentry{
    char *myname;                     /* print name of symbol          */
    struct Isymtabentry *nextsymbol;  /* ptr to next entry             */
    struct symboltable *mytable;      /* ptr to symbol table structure */
    struct tnodev *mynode;            /* ptr to declaration of symbol  */
    int refcount;                     /* reference count               */
    int value;                        /* value of integer variable     */
    };

/* This is the only symbol table routine taken from standard that must be
   replaced.  The code is identical to the default routine but, since the
   size of the symbol table entry is larger, it must be recompiled in
   order to get the correct size for the allocation.  In addition, some
   of the fields have been casted in order to get the correct typing.  */

struct symtabentry *newstentry(thestable,name)
struct symboltable *thestable;
char *name;
{
    struct Isymtabentry *retentry;      /* ptr to the new entry */

    retentry = Isymtab(malloc(sizeof(*retentry)));
    retentry->myname = newstring(name);
    strcpy(retentry->myname,name);
    retentry->nextsymbol = Isymtab(thestable->entries);
    retentry->mytable = thestable;
    retentry->refcount = 1;
    thestable->entries = Symtab(retentry);
    return Symtab(retentry);
}

/* This routine implements a standard expression evaluation routine.  For
   integer constants the ASCII value (the way ALOE stores constants) is
   converted and returned.  Variable values are taken from the symbol
   table.  Plus and times are evaluated by recursive calls. */

int evalexp(node)
struct tnode *node;
{
    switch(node->opt) {
        case iINT:
            return (atoi(Tnodec(node)->ctname));
        case iLOOPVAR:
            return (Isymtab(Tnodev(node)->key)->value);
        case iPLUS:
            return (evalexp(getson(node,PLUSexp1)) +
                    evalexp(getson(node,PLUSexp2)));
        case iTIMES:
            return (evalexp(getson(node,TIMESexp1)) *
                    evalexp(getson(node,TIMESexp2)));
        default:
            return (0);
    }
}
```

```
/* This routine does the actual interpretation of the statements.  It is
   passed a list of statements to execute in order.  Print statements
   go through each expression in the list printing out the evaluated
   value.  For loops it controls the bounds and deals with nesting
   through recursion.  */

struct tnode *cmdRUN(thisnode)
struct tnode *thisnode;
{
        struct listnode *curstmt,       /* header for statement being
                                           interpreted               */
                        *curelem;       /* header for element being
                                           printed                   */
        struct tnode *stmt;             /* statement being executed  */
        struct tnodev *loopvar;         /* the loop variable's node  */
        int step, endvalue;             /* loop control variables    */

    /* loop through each statement */

    for (curstmt = getlist(thisnode); curstmt != NIL; curstmt = cdr(curstmt)) {
        stmt = getcar(curstmt);
        if (stmt->opt == iPRINT)
            for (curelem = getlist(stmt); curelem != NIL; curelem = cdr(curelem))
                printf("%9d",evalexp(getcar(curelem)));
        else if (stmt->opt == iFOR) {
            if (getson(stmt,FORstepexp)->opt == iEMPTYSTEP)
                step = 1;
            else
                step = evalexp(getson(stmt,FORstepexp));
            loopvar = Tnodev(getson(stmt,FORloopvar));
            endvalue = evalexp(getson(stmt,FORtoexp));
            for (Isymtab(loopvar->key)->value =
                         evalexp(getson(stmt,FORfromexp));
                 Isymtab(loopvar->key)->value <= endvalue;
                 Isymtab(loopvar->key)->value += step)
                 cmdRUN(getson(stmt,FORstmts));
        }
    }
}

/* Implementation of extended command.  It just starts the interpretation
   at the root of the program.  */

struct tnode *exRUN(thisnode)
struct tnode *thisnode;
{
        setoutwind("user",1);
        cmdRUN(getson(getsysroot(),PROGRAMstmts));
        resetoutwind();
        return thisnode;
}

/* The extended command table describing the one added command, its
   synonym, its defining function, and permission to execute it
   initially. */

struct edcommands extcomtable[] =
{
        "RUN","\001\022",exRUN,1,               /* ↑a↑r          */
        0,0,0,0
};
```

```
/* The action routine for INT that checks to see that the constant
   is numeric.  If it is not, the value is replaced by a 0 and a
   warning is passed to the user.  */

aINT(thisnode, actkind)
struct tnode *thisnode;
int actkind;
{
    char *digit;

    switch (actkind) {
        case CREATE:
            for (digit = Tnodec(thisnode)->ctname; *digit != '\0'; ++digit)
                if (*digit < '0' || '9' < *digit)
                    break;
            if (*digit != '\0') {
                changedtree();
                error1("%s is non-numeric",thisnode,Tnodec(thisnode)->ctname);
                return(-2);     /* abort the creation */
            }
            break;
        default:
            break;
    }
    return NIL;
}
```

References

[Ada 80] United States Department of Defense.
Reference Manual for the Ada Programming Language.
1980.
Proposed Standard Document.

[Ball 80] Ball, J. E.
Alto as Terminal.
1980.
Carnegie-Mellon University.

[Feiler 79] Feiler, P. H. and Medina-Mora, R.
The GC Language.
1979.
Gandalf Internal Documentation. Carnegie-Mellon University.

[Feiler 81] Feiler, P. H.
The Gandalf Display Package.
1981.
Gandalf Internal Documentation. Carnegie-Mellon University.

[Gosling 81a] Gosling, J.
Unix Emacs
Carnegie-Mellon University, 1981.

[Gosling 81b] Gosling, J.
A Redisplay Algorithm.
In *Proceedings of the ACM SIGPLAN/SIGOA Symposium on Text Manipulation.* ACM SIGPLAN/SIGOA, June, 1981.

[Habermann 79] Habermann, A. Nico.
The Gandalf Research Project.
In *Computer Science Research Review 1978-79*, pages 28-35. Carnegie-Mellon University, 1979.

[Habermann 80] Habermann, A. Nico.
Notes on Programatics and its Language Alfa.
1980.
Private Communication.

[Jensen 74] Jensen, K. and Wirth, N.
Pascal User Manual and Report.
Springer-Verlag, 1974.

[Kernighan 78] Kernighan, B. W. and Ritchie, D. M.
Prentice-Hall Software Series: The C Programming Language.
Prentice-Hall, 1978.

[Notkin 82]	Notkin, D. S. and Kaiser, G. E. The Implementation of the Gandalf Software Development Environment. 1982. Carnegie-Mellon University. To appear.

Index

DATE
ILME